Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

Забезпечення відмовостійкості та доступності серверів ліцензування з апаратними ключами в контейнеризованому середовищі (Docker/Kubernetes)

ПОЯСНЮВАЛЬНА ЗАПИСКА

Виконав студент 2 курсу, групи 2КІ—24м
спеціальності 123 — Комп'ютерна інженерія
Ясько Я. М.
Керівник к.т.н., проф. каф. ОТ
Азарова А. О.
"15" 12 2025 р.
Опонент к.т.н., доц. каф. МБІС
Карпінець В. В.
"15" 12 2025 р.

Допущено до захисту
зав каф. ОТ
д.т.н., проф. Азаров О.Д.
"18" 12 2025 р.

ВНТУ 2025

ВІННИЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки
Освітній рівень — магістр
Спеціальність — 123 Комп'ютерна інженерія

ЗАТВЕРДЖУЮ

Завідувач кафедри
обчислювальної техніки
докт. техн. наук, професор
_____Олексій АЗАРОВ
"25" вересня 2025 р.

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ

студенту **Яську Якову Михайловичу**

1 Тема роботи «Забезпечення відмовостійкості та доступності серверів ліцензування з апаратними ключами в контейнеризованому середовищі (Docker/Kubernetes)» керівник роботи Азарова Анжеліка Олексіївна к.т.н., професор, затверджено наказом вищого навчального закладу від 15.09.2025 року № 313.

2 Строк подання студентом роботи **5.12.2025**

3 Вихідні дані до роботи: середовище розробки Visual Studio, мова програмування C#, технічний опис застосунку, статистичні дані.

4 Зміст розрахунково—пояснювальної записки (перелік питань, які потрібно розробити): вступ, аналіз сучасних систем ліцензування з використанням апаратних ключів, розроблення апаратної частини методу забезпечення відмовостійкості та доступності серверів ліцензування, реалізація програмної частини методу та тестування розробленого рішення, економічне обгрунтування та оцінювання ефективності запропонованого рішення.

5 Перелік графічного матеріалу (з точним зазначенням обов'язк
креслень): Схема апаратного ключа, GPS модуля, лыстинг програми апарат
ключа, лістинг програми медичного gateway.

6 Консультанти розділів роботи приведені в табл. 1.

Таблиця 1— Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1 – 4	Азарова А. О. к.т.н., професор каф. ОТ		
5	Ратушняк М. І. к.т.н., доц каф. ЕПВМ		
Нормоконт-роль	Швець С. І. асист. каф. ОТ		

7 Дата видачі завдання 30.10.2025

8 Календарний план виконання МКР наведено у табл. 2.

Таблиця 2 — Календарний план

№ з/п	Назва етапів МКР	Строк виконання	Підпи
1	Постановка задачі	12.09.2025	
2	Аналіз сучасного стану досліджень	15.09.2025	
3	Огляд існуючих рішень	26.09.2025	
4	Аналіз праць видатних дослідників	04.10.2025	
5	Пошук та аналіз матеріалів для розрахунків	14.10.2025	
6	Покращення використовуваного методу	21.10.2025	
7	Розробка та тестування програми	02.11.2025	
8	Розрахунок економічної частини	14.11.2025	
9	Оформлення пояснювальної записки та ілюстративного матеріалу	16.11.2025	
10	Виконання магістерської кваліфікаційної роботи	28.11.2025	
11	Перевірка якості виконання магістерської кваліфікаційної роботи та усунення недоліків	09.11.2025	
12	Підписи супроводжувальних документів у керівника, опонента, нормоконтролера	16.12.2025	

Студент _____ Яків ЯСЬКО

Керівник _____ канд. техн. наук, професор каф. ОТ Анжеліка АЗАРОВА

АНОТАЦІЯ

УДК 004.4

Яcько Я. М. Забезпечення відмовостійкості та доступності серверів ліцензування з апаратними ключами в контейнеризованому середовищі (Docker/Kubernetes). Магістерська кваліфікаційна робота, спец. 123 комп'ютерна інженерія. Вінниця: ВНТУ, 2025 — 90 с. На укр. мові. Бібліогр.: 44 назв; рис.: 21; табл. 1.

У роботі досліджено метод забезпечення відмовостійкості та високої доступності серверів ліцензування, що використовують апаратні ключі. Проаналізовано сучасні підходи до ліцензування ПЗ та апаратних рішень (USB—dongle, HSM, TPM), визначено їхні недоліки та обмеження. Розроблено апаратну та програмну складові методу, що забезпечують захист апаратних ключів, резервування, георепліkацію та автоматичне відновлення у кластерному середовищі Docker/Kubernetes. Проведені експерименти підтвердили ефективність та стійкість рішення до відмов контейнерів, вузлів і втрати обладнання. Виконано економічне обґрунтування впровадження та оцінено вартість забезпечення відмовостійкості.

Ключові слова: відмовостійкість, ліцензування, апаратний ключ, USB—dongle, HSM, TPM, Kubernetes, Docker, кластер, висока доступність, контейнеризація.

ABSTRACT

UDC 004.4

Yasko Y. M. Ensuring Fault Tolerance and Availability of Licensing Servers with Hardware Keys in a Containerized Environment (Docker/Kubernetes). Master's Qualification Thesis, Specialty 123 — Computer Engineering. Vinnytsia: Vinnytsia National Technical University (VNTU), 2025. 90 pp. In Ukrainian. Bibliography: 44 references; Figures: 21; Tables: 1.

The thesis explores a method for ensuring fault tolerance and high availability of licensing servers that rely on hardware security keys. Modern software licensing systems and hardware—based protection solutions (USB dongles, HSM, TPM) are analyzed, and their limitations and challenges are identified. A hardware and software solution is developed to provide protection of hardware keys, redundancy, geo—replication, and automatic recovery within a Docker/Kubernetes cluster. Experimental testing confirmed the effectiveness and resilience of the proposed approach under various failure scenarios, including container crashes, node failures, and hardware key loss. An economic justification of the method is performed, along with an evaluation of the cost of providing fault tolerance.

Keywords: fault tolerance, licensing, hardware key, USB dongle, HSM, TPM, Kubernetes, Docker, cluster, high availability, containerization.

ЗМІСТ

ВСТУП

Розвиток інформаційних технологій, поширення програмного забезпечення у різних сферах діяльності людини та необхідність його надійного захисту зумовили появу систем ліцензування. Ще у 1980—1990—х роках розробники почали активно впроваджувати апаратні ключі, які запобігали несанкціонованому копіюванню та використанню програмних продуктів. Такі рішення широко застосовувалися у складних інженерних системах — CAD/CAM/CAE, GIS, медичних, наукових і промислових програмних комплексах. Із розвитком мережевих технологій і переходом до хмарних сервісів виникла потреба у забезпеченні відмовостійкості та безперервності роботи ліцензійних серверів. Якщо раніше достатньо було гарантувати локальне зберігання апаратного ключа, то сьогодні критично важливою стала здатність системи зберігати працездатність навіть у разі збоїв, втрати вузла або мережевого порушення. Це питання особливо актуальне у сучасних інфраструктурах, де активно використовуються технології віртуалізації, контейнеризації та оркестрації.

Попри значний прогрес у розвитку контейнерних технологій, інтеграція апаратних ключів у такі середовища залишається складним завданням. Фізична прив’язка ключа до конкретного вузла, складність масштабування та проблеми з доступом у разі відмови серверів обмежують надійність таких систем. Існуючі рішення часто є дорогими, закритими або не забезпечують автоматичного відновлення у кластерних середовищах. Отже, актуальним є розроблення нового методу забезпечення відмовостійкості та доступності серверів ліцензування, який би поєднував переваги апаратного захисту з гнучкістю сучасних контейнерних технологій.

Мета дослідження магістерської кваліфікаційної роботи полягає у розроблені методу забезпеченні відмовостійкості та високої доступності серверів ліцензування з апаратними ключами в контейнеризованому середовищі, допоможе поєднути високу доступність системи з високим рівнем захисту.

Для досягнення поставленої мети було поставлено та вирішено такі завдання:

— проаналізовано існуючі системи ліцензування з апаратними ключами та визначено їхні недоліки у контексті відмовостійкості;

— досліджено переваги та недоліки існуючих підходів до побудови відмовостійких серверних систем;

— розроблено апаратну частину методу, що забезпечує фізичний і криптографічний захист ключів;

— реалізовано програмну частину, що уможливлює доступ до апаратного ключа у контейнеризованому середовищі;

— здійснено тестування і експериментальну перевірку працездатності методу забезпечення відмовостійкості;

— доведено економічну доцільність впровадження запропонованого рішення.

Об’єктом дослідження є процес забезпечення відмовостійкості та доступності серверів ліцензування з апаратними ключами в контейнеризованому середовищі

Предмет дослідження магістерської кваліфікаційної роботи — метод забезпечення відмовостійкості та доступності серверів ліцензування з апаратними ключами в контейнеризованому середовищі

Серед **методів дослідження**, використаних у магістерській роботі є загальнонаукові та специфічні, зокрема:

— метод об’єктно—орієнтованого програмування застосовано для розроблення надійної комп’ютеризованої системи для тестування апаратного ключа;

— метод хмарних обчислень для аналізу відмовостійкої архітектури в контейнеризованому та кластерному середовищі (Kubernetes, контейнеризація, оркестрація, балансування навантаження); метод дотримання інформаційної безпеки із застосуванням принципів Zero Trust, RBAC, mTLS, Secret Management, щоб гарантувати захищений доступ до апаратних ключів та ліцензійних сервісів;

— метод експерименту для перевірки працездатності сервісу, коректності алгоритму доступу, відстеження журналів подій, оцінювання поведінки системи за різних конфігурацій апаратних ключів.

Наукова новизна отриманих результатів полягає в удосконаленні методу забезпечення відмовостійкості та доступності серверів ліцензування, що, на відміну від існуючих підходів, із використанням апаратних ключів дозволяє підвищити надійність роботи ліцензійного сервера без втручання оператора.

Практичне значення отриманих результатів полягає у можливості впровадження розробленого методу у корпоративні IT—системи, що використовують ліцензування на базі апаратних ключів. Рішення забезпечує безперервність роботи критично важливих сервісів, скорочує час простою, підвищує безпеку зберігання ліцензійних даних і може бути інтегроване у хмарні або локальні середовища з мінімальними витратами на адаптацію.

Публікації. Результати проведених досліджень було опубліковано у розділі колективної монографії [22], а також тезах доповідей на міжнародній конференції [23] та захищено двома свідоцтвами на реєстрацію авторського права на комп'ютерні програми [24, 25].

Апробацію результатів було здійснено на Міжнародній науково-технічній Інтернет-конференції ВНТУ «Молодь в науці та освіті» (м. Вінниця, 2025) [Електронний ресурс]. Режим доступу: https://conferences.vntu.edu.ua/index.php/mn/mn2026/paper/view/26808 [23].

1 АНАЛІЗ СУЧАСНИХ СИСТЕМ ЛІЦЕНЗУВАННЯ З ВИКОРИСТАННЯМ АПАРАТНИХ КЛЮЧІВ

1.1 Роль та значення ліцензування у сучасних програмних продуктах

У сучасному світі програмне забезпечення стало одним із найважливіших ресурсів, що визначають розвиток технологій, бізнесу, науки та освіти. Його створення потребує значних інтелектуальних і фінансових витрат, тому захист результатів програмної діяльності є одним із ключових завдань розробників і компаній, які постачають ПЗ на ринок. У цьому контексті ліцензування програмного забезпечення виступає не лише юридичним інструментом регулювання відносин між розробником і користувачем, а й технічним механізмом, що забезпечує контроль за використанням програмного продукту.

Поняття «ліцензія» в контексті програмного забезпечення означає надання користувачеві певних прав на встановлення, використання або розповсюдження програми згідно з умовами розробника чи власника авторських прав. Ліцензійна угода визначає кількість дозволених інсталяцій, тип доступу (персональний, корпоративний, мережевий), термін дії, а також функціональні можливості, що залежать від рівня підписки або купленої редакції. Саме через механізми ліцензування виробник має змогу контролювати обсяг використання ПЗ, захищати свій продукт від несанкціонованого копіювання, а також отримувати дохід, необхідний для подальшого розвитку [1].

Значення ліцензування особливо зростає у галузях, де програмне забезпечення є не просто інструментом, а критичною складовою виробничих процесів. Це системи автоматизованого проєктування (CAD/CAM/CAE), інженерні комплекси, геоінформаційні системи, фінансові застосунки, медичне та наукове ПЗ. Такі продукти, як правило, мають високу вартість і потребують спеціального контролю за доступом до функціоналу. Відсутність належного захисту призводить до значних економічних втрат розробників через піратство та нелегальне використання ліцензій.

Історично перші системи ліцензування реалізовувалися у вигляді програмних ключів — унікальних кодів, що активували програму на конкретному комп’ютері. Проте з поширенням глобальних мереж та віртуалізації цей підхід виявився недостатньо безпечним: ключі можна було копіювати або зламувати. У відповідь на ці виклики були створені апаратні засоби ліцензування, або апаратні ключі (рис. 1.1). Вони забезпечують апаратний рівень захисту, що робить несанкціоноване копіювання програмного забезпечення значно складнішим або практично неможливим.

Рисунок 1.1 — Апаратний ключ

Апаратний ключ виконує роль фізичного носія ліцензії — він містить зашифровану інформацію про права користувача, унікальний ідентифікатор, криптографічні ключі та алгоритми перевірки автентичності. Такий пристрій зазвичай підключається до комп’ютера через USB—інтерфейс або інтегрується у локальну мережу як ліцензійний сервер. Програма звертається до ключа при запуску або виконанні певних функцій, перевіряючи наявність і дійсність ліцензії. У разі відсутності ключа або невідповідності даних виконання програми блокується.

Із технічного погляду, сучасні системи ліцензування з апаратними ключами виконують кілька важливих функцій:

—ідентифікація користувача або робочого місця кожен ключ має унікальний ідентифікатор;

—контроль кількості активних ліцензій для мережевих продуктів обмежується кількість одночасних підключень;

—захист від підроблення або емуляції за рахунок криптографічних протоколів та апаратного шифрування;

—автоматичне відключення при порушеннях, наприклад, під час спроби перенесення ключа без авторизації.

Таким чином, ліцензування виконує подвійну функцію економічну (захист комерційних інтересів розробника) та технічну (забезпечення безпеки програмного продукту). Його роль стає ще важливішою у корпоративних середовищах, де від безперервності роботи ліцензійних серверів залежить функціонування багатьох бізнес—процесів.

В умовах глобальної цифровізації та поширення контейнеризованих інфраструктур виникає потреба у нових підходах до організації ліцензування. Сучасні компанії прагнуть інтегрувати апаратні ключі у віртуальні середовища, розподілені кластери та хмарні платформи. Це відкриває нові можливості, але водночас створює низку проблем, пов’язаних із відмовостійкістю, масштабованістю та безпекою доступу до фізичних пристроїв [2].

Отже, ліцензування у сучасних програмних продуктах є фундаментальним механізмом не лише захисту авторських прав, а й забезпечення стабільної, контрольованої та безпечної експлуатації програмних систем. Його розвиток безпосередньо впливає на економічну ефективність, надійність і конкурентоспроможність програмних рішень, а тому дослідження методів удосконалення ліцензування, зокрема в контейнеризованих середовищах, є актуальним завданням сучасної інформаційної інженерії.

Забезпечення відмовостійкості та доступності сервісів у контейнеризованих середовищах є важливою задачею сучасної інженерії розподілених систем. У випадку серверів ліцензування, що використовують апаратні ключі, складність зростає через наявність унікального фізичного

ресурсу, який не може бути масштабований стандартними механізмами Kubernetes. Тому для таких систем актуальними є підходи, що поєднують кластеризацію, контроль доступу, серіалізацію запитів, ідемпотентність, відновлення після збоїв та моніторинг надійності, а також інженерні практики експлуатації високонавантажених систем.

Проаналізовані джерела підтверджують, що для контейнеризованих середовищ ключовими засобами підвищення доступності є автоматичне відновлення компонентів, коректне керування станом, продумана архітектура взаємодії сервісів та наявність експлуатаційних практик моніторингу й оцінювання надійності. Однак для серверів ліцензування з апаратними ключами вирішальним фактором є обмеження унікального фізичного ресурсу, що вимагає централізованого контролю доступу (наприклад, через API—шлюз), серіалізації запитів, ідемпотентності та узгодженого механізму відновлення у разі збоїв.

1.2 Аналіз переваг і недоліків систем ліцензування з апаратними засобами захисту

Системи ліцензування з використанням апаратних засобів захисту є одним із найефективніших інструментів протидії несанкціонованому використанню програмного забезпечення. Їхня поява стала логічним етапом еволюції засобів захисту авторських прав — від простих програмних ключів і серійних номерів до апаратних пристроїв, що реалізують складні криптографічні алгоритми. Такі рішення використовуються в багатьох комерційних, інженерних та промислових продуктах, де втрати від піратства можуть перевищувати витрати на впровадження захисних технологій.

Гаспар П., Сімоеш П. та ін. у роботі [3] розглядають процес міграції контейнеризованих застосунків до cloud-native середовищ з метою підвищення доступності, масштабованості та безпеки інформаційних систем. Автори аналізують архітектурні особливості Kubernetes та демонструють експериментальну реалізацію на базі хмарної платформи Azure. Отримані результати підтверджують ефективність контейнерної оркестрації для

забезпечення високої доступності сервісів. Разом із тим, у роботі не розглядається проблема доступу до фізично унікальних апаратних ресурсів, що обмежує застосування запропонованого підходу для систем ліцензування з апаратними ключами.

Хе Ї., Білал К. у роботі [4] пропонують стратегію оркестрації Kubernetes-кластерів з урахуванням поточного навантаження, спрямовану на оптимізацію використання ресурсів у спільно використовуваних середовищах. Дослідження демонструє, що динамічне прийняття рішень щодо розміщення подів дозволяє підвищити продуктивність мікросервісних застосунків. Водночас запропонований підхід орієнтований переважно на балансування обчислювальних ресурсів і не враховує обмеження, пов’язані з ексклюзивним доступом до апаратних криптографічних модулів.

Кавіта С., Прабу П. та ін. у статті [5] представляють безпечну та відмовостійку систему пошуку медичних зображень у розподілених хмарних середовищах. Автори приділяють особливу увагу усуненню єдиної точки відмови та забезпеченню безперервної доступності медичних даних. Робота є показовою з точки зору вимог до медичних інформаційних систем, однак у ній не розглядаються питання апаратного захисту криптографічних операцій та ліцензування програмного забезпечення.

Тасич Д., Євтич Д. та ін. у роботі [6] здійснюють оцінювання продуктивності архітектур ARM та RISC-V для високопродуктивних обчислень із використанням Docker і Kubernetes. Дослідження підтверджує можливість ефективного застосування ARM-платформ у контейнеризованих середовищах. Разом із тим, автори зосереджуються переважно на обчислювальних характеристиках і не аналізують аспекти безпеки та контролю доступу до апаратних криптографічних пристроїв.

Раут Р. К., Дас С. К. та ін. у роботі [7] пропонують відмовостійку мікросервісну архітектуру з адаптивним балансуванням навантаження на основі Docker і Spring Cloud. Автори експериментально демонструють зменшення часу відновлення після відмов та підвищення пропускної здатності системи.

Недоліком підходу є орієнтація на програмні сервіси, що можуть масштабуватися горизонтально, без урахування наявності фізично унікальних ресурсів, таких як апаратні ключі.

Вібу-Системс АГ у публікації [8] описують практичні аспекти інтеграції сервера ліцензування CodeMeter у середовище Kubernetes, зокрема використання апаратних ключів та особливості забезпечення доступності ліцензій у контейнеризованих розгортаннях. Матеріал має значну практичну цінність і ґрунтується на реальному промисловому досвіді впровадження комерційного рішення. Водночас розглянутий підхід значною мірою прив’язаний до конкретної екосистеми CodeMeter та пропонує фіксований набір архітектурних рішень, що обмежує можливість його адаптації або узагальнення для інших сценаріїв використання апаратних ключів і альтернативних моделей ліцензування в контейнеризованих середовищах.

Отже, аналіз сучасних наукових і практичних підходів показав, що, попри значний прогрес у сфері забезпечення високої доступності та відмовостійкості контейнеризованих систем, наявні рішення не враховують у повній мірі специфіку серверів ліцензування з апаратними ключами як фізично унікальних ресурсів. Недостатня увага до механізмів ексклюзивного доступу, серіалізації криптографічних операцій та коректного відновлення роботи у разі відмов окремих вузлів зумовлює необхідність подальшого розвитку методів забезпечення відмовостійкості таких серверів. Саме розв’язання зазначених проблем і є основною задачею даної магістерської кваліфікаційної роботи.

Основна ідея апаратного захисту полягає в тому, що для запуску або роботи програмного забезпечення необхідна фізична наявність спеціального пристрою — апаратного ключа. Цей ключ, як правило, підключається через інтерфейс USB або використовується як мережевий модуль, який зберігає зашифровану інформацію про ліцензію, криптографічні ключі та алгоритми перевірки автентичності. Програма взаємодіє з ним за допомогою спеціального API або драйвера, виконуючи перевірку перед початком роботи.

Переваги систем ліцензування з апаратними засобами захисту:

— високий рівень безпеки , на відміну від програмних ключів, які можуть бути скопійовані або зламані, апаратний ключ містить криптографічно захищену інформацію, до якої неможливо отримати доступ без фізичного пристрою, більшість сучасних ключів підтримують шифрування з використанням алгоритмів AES, RSA, ECC, а також реалізують механізми захисту від реверс—інжинірингу та апаратного злому (апарат виявлення, самознищення даних при спробі зламу тощо);

— надійна ідентифікація користувача, кожен ключ має унікальний ідентифікатор, що дозволяє точно визначати власника ліцензії, кількість активних копій програми та права доступу. це особливо важливо для корпоративних і мережевих середовищ, де контроль за кількістю одночасних користувачів є обов’язковим;

— захист від підроблення та емуляції, апаратні ключі використовують унікальні криптографічні обчислення (наприклад, challenge—response), що унеможливлює створення програмного аналога, навіть у разі копіювання програмного коду, відсутність справжнього ключа блокує роботу ПЗ;

— незалежність від мережевих збоїв або серверів авторизації, у локальному режимі роботи апаратний ключ забезпечує автономну перевірку ліцензії без потреби постійного з’єднання з сервером, що знижує ризики зупинки програмного продукту через проблеми з мережею;

— довготривала експлуатація, більшість сучасних апаратних ключів мають термін служби понад 10 років, не потребують складного обслуговування і сумісні з різними операційними системами.

Недоліки систем ліцензування з апаратними засобами захисту:

— залежність від фізичного пристрою, основною проблемою є необхідність фізичного підключення ключа, у разі його пошкодження, втрати або виходу з ладу користувач може повністю втратити доступ до програмного забезпечення, доки не буде отримано заміну, це створює ризики простою, особливо у випадку віддалених або критично важливих систем;

— складність інтеграції у контейнеризовані та хмарні середовища, апаратні ключі розроблялися для традиційних фізичних серверів або локальних робочих станцій, у контейнерах або віртуальних машинах їх підключення потребує спеціальних механізмів, таких як USB—over—IP або емуляція пристроїв, що часто призводить до проблем з продуктивністю, безпекою чи сумісністю [9];

— обмежена масштабованість, на відміну від програмних ліцензій, які можна легко масштабувати шляхом активації додаткових ключів у базі даних, апаратні рішення вимагають наявності фізичних пристроїв, це ускладнює швидке розширення інфраструктури або міграцію на інші платформи;

— висока вартість впровадження, вартість апаратних ключів, драйверів і серверів ліцензій є суттєвою, для великих організацій із сотнями користувачів це може призвести до значних початкових витрат (CAPEX), а також витрат на підтримку (OPEX). Необхідність резервування та управління.

Для забезпечення відмовостійкості потрібне дублювання ключів або побудова кластерних ліцензійних серверів, що вимагає додаткової інфраструктури, моніторингу та адміністрування [10].

Таким чином, системи ліцензування з апаратними засобами захисту мають високий рівень безпеки, стабільність та надійність, що робить їх оптимальними для критично важливих застосунків і комерційного програмного забезпечення з високою вартістю. Проте їх основними недоліками залишаються залежність від фізичного пристрою, складність інтеграції у сучасні контейнеризовані інфраструктури та висока вартість володіння [11].

Ці фактори зумовлюють актуальність подальших досліджень і розроблення методів підвищення відмовостійкості та доступності серверів ліцензування, які поєднували б переваги апаратного захисту з гнучкістю сучасних хмарних і контейнерних технологій.

1.3 Дослідження архітектурних підходів до побудови відмовостійких сервісів із використанням апаратних ключів

Із розвитком інформаційних технологій, хмарних платформ і контейнеризації особливої актуальності набули питання забезпечення відмовостійкості та високої доступності серверних систем. Для ліцензійних серверів, що використовують апаратні ключі як засіб автентифікації та контролю доступу до програмного забезпечення, ці завдання мають специфічний характер, оскільки такі ключі є фізичними пристроями, які мають обмеження у масштабуванні, переміщенні й дублюванні. Забезпечення безперервної роботи подібних систем потребує комплексного підходу, який враховує як програмну, так і апаратну складові [12].

Традиційно найпростішим способом побудови ліцензійного сервера є локальне підключення апаратного ключа до фізичного вузла, де виконується програма—сервер. Така архітектура має мінімальні затримки, не потребує складних мережевих налаштувань і є ефективною для невеликих підприємств або лабораторій. Проте її головним недоліком є наявність єдиної точки відмови: у разі виходу з ладу сервера або самого ключа вся система стає недоступною. Для великих корпоративних середовищ це неприпустимо, тому виникла потреба у розподілених архітектурних рішеннях [13].

Одним із підходів є використання мережевих апаратних ключів, які функціонують як окремі сервери ліцензій. У цьому випадку кілька користувачів або застосунків можуть одночасно підключатися до спільного ключа через локальну мережу. Така архітектура дозволяє централізовано керувати ліцензіями, відслідковувати їхнє використання та забезпечувати контроль доступу на рівні користувачів. Водночас вона також створює ризики — у разі відмови вузла з ключем або мережевого інтерфейсу система втрачає доступ до ліцензій, що може призвести до зупинки критичних процесів. Для мінімізації таких ризиків застосовуються механізми резервування, дублювання серверів і використання балансувальників навантаження.

Сучасні архітектури з високою доступністю (High Availability) реалізуються за схемами active—passive або active—active (рис. 1.2). У першому випадку основний сервер має ексклюзивний доступ до апаратного ключа, тоді як

резервний перебуває у режимі очікування. У разі збою система автоматично перемикається на резервний вузол. Такий підхід забезпечує передбачувану поведінку системи, однак потребує коректної синхронізації станів і механізмів “fencing”, щоб уникнути ситуацій, коли два сервери одночасно намагаються звертатися до одного ключа.

У схемі active—active декілька серверів або контейнерів можуть одночасно обслуговувати клієнтів, але звернення до апаратного ключа відбувається через спеціальний програмний шлюз або сервіс—посередник. Цей шлюз виконує серіалізацію запитів, забезпечує контроль черги, а також веде журнал операцій видачі й повернення ліцензій.

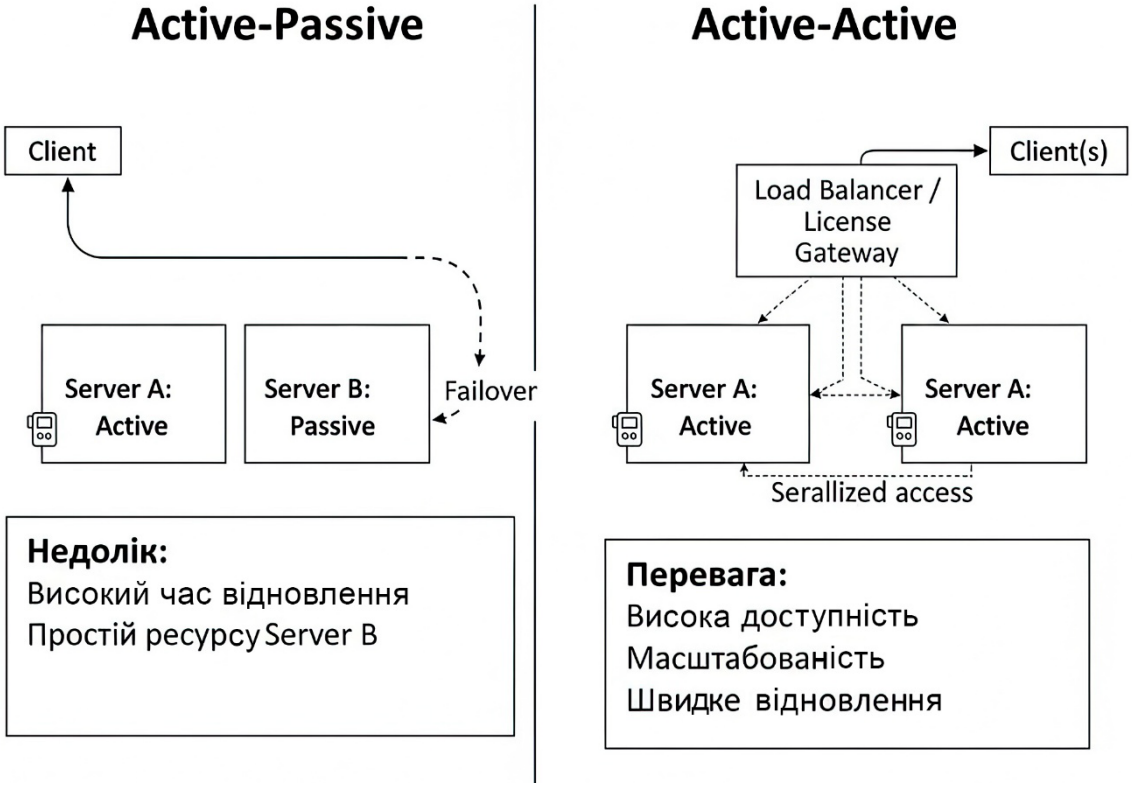


Рисунок 1.2 — Схема взаємодії active—active, active—passive

Така модель більш гнучка, вона дозволяє масштабувати клієнтське навантаження, однак вимагає особливої уваги до надійності самого шлюзу, який стає критичним елементом системи.

В умовах контейнеризації, коли сервіси розгортаються у середовищах на

зразок Kubernetes, з’являються додаткові інструменти для реалізації відмовостійкості. Для підключення апаратних ключів у таких середовищах використовують Device Plugin (рис. 1.3) — спеціальний компонент, який інформує оркестратор про наявність доступного апаратного ресурсу. Завдяки цьому можна забезпечити, щоб контейнери, які потребують доступу до ключа, розміщувалися виключно на вузлах, де такий ключ підключено. Додаткові механізми, як—от PodDisruptionBudget, допомагають обмежити кількість одночасно недоступних контейнерів під час оновлень або перезапусків, а Topology Spread Constraints забезпечують рівномірний розподіл реплік по вузлах кластера, що знижує ймовірність одночасного збою.

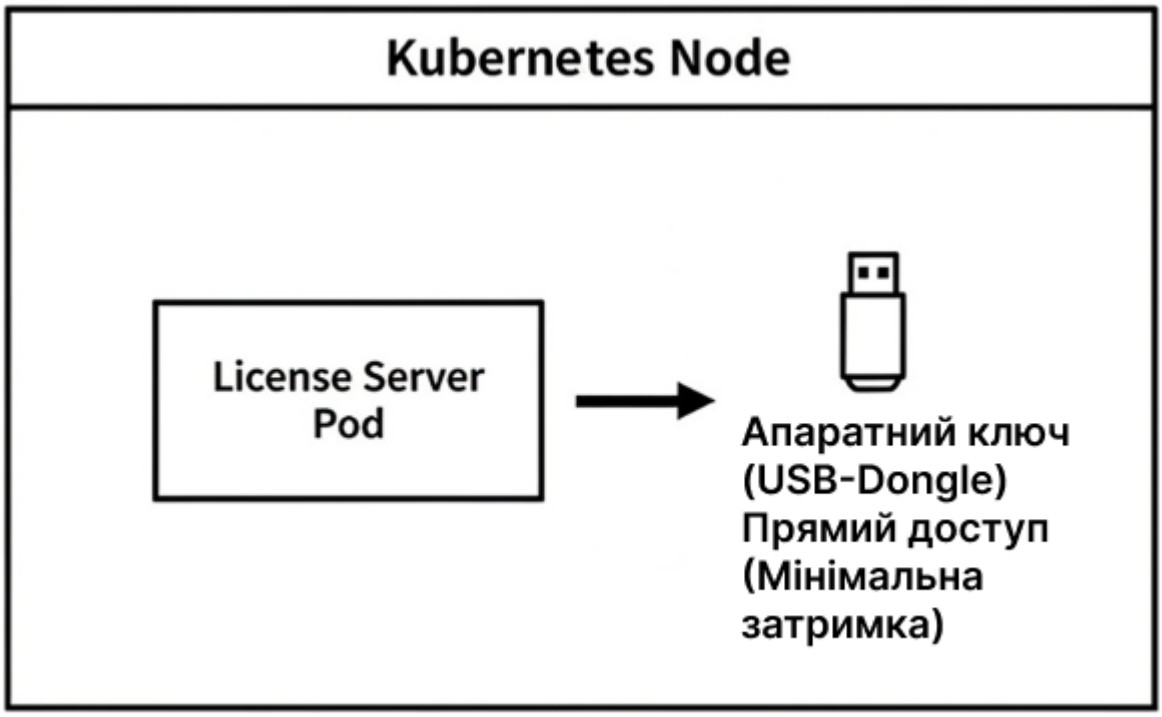


Рисунок 1.3 — Схема підключення Device Plugin у кластер Kubernetes

Особливу роль у побудові відмовостійких систем відіграють health—checks та watchdog—сервіси, які постійно перевіряють стан апаратного ключа, драйвера та ліцензійного демона. У разі втрати зв’язку або збоїв вони ініціюють автоматичне перезапускання контейнера або перемикання на резервний вузол. У поєднанні з механізмами leader election це дозволяє досягти автоматичного відновлення працездатності без втручання адміністратора [15].

Важливим аспектом також є безпека. Оскільки апаратний ключ є джерелом істини для ліцензійної системи, доступ до нього має бути захищений на всіх рівнях — від фізичного до мережевого. Для цього використовуються протоколи взаємної автентифікації (mTLS), ролева модель доступу (RBAC), ізольовані мережеві простори (NetworkPolicy), а також секрет—менеджери, які відповідають за безпечне зберігання криптографічних параметрів. Таким чином, забезпечується як цілісність процесу ліцензування, так і захист від внутрішніх і зовнішніх загроз.

У розподілених інфраструктурах, де доступ до апаратних ключів здійснюється через мережу, важливим завданням є мінімізація затримок і забезпечення стабільного каналу зв'язку. Для цього застосовуються технології VPN, Zero—Trust Network Access, а також системи моніторингу, що відстежують продуктивність ліцензійних запитів, рівень помилок та час відгуку (latency). За допомогою метрик типу p95 або p99 адміністратори можуть вчасно виявляти деградацію продуктивності та запобігати аваріям.

Також одним із ключових аспектів побудови відмовостійких сервісів є забезпечення високої доступності системи зберігання даних. Оскільки апаратний ключ є лише одним із компонентів комплексної інфраструктури захисту, стабільність роботи програмного сервісу значною мірою залежить від доступності бази даних, у якій зберігаються метадані ліцензій, журнали аудиту та конфігураційні параметри. У разі недоступності бази втрачається можливість фіксації операцій, виконання перевірок ліцензій або обробки транзакцій, що може призвести до блокування сервісу навіть за наявності справного апаратного ключа. Тому архітектура системи повинна враховувати механізми реплікації, автоматичного перемикання між вузлами та розподілення навантаження [14].

Залежно від вимог до узгодженості даних, продуктивності та масштабованості можуть застосовуватися різні типи баз даних. Традиційні реляційні системи керування базами даних (PostgreSQL, MySQL, SQL Server) забезпечують високу цілісність і транзакційність, що є критичним для ліцензійних операцій, де неприпустимі дублікати або некоректні стани. Для

забезпечення доступності ці системи підтримують синхронну та асинхронну реплікацію, режим primary—standby, а також кворумні кластери на основі групової реплікації. Нереляційні бази даних (MongoDB, Cassandra, Redis), натомість, краще підходять для високонавантажених і географічно розподілених систем, де важливо мінімізувати латентність і забезпечити горизонтальне масштабування.

У кластерних середовищах із використанням апаратних ключів доцільно комбінувати кілька типів сховищ залежно від задачі: реляційні СУБД використовуються для зберігання критичних параметрів ліцензування та аудиту, тоді як нереляційні сховища можуть застосовуватись для кешування частих запитів, зниження навантаження на основну базу та прискорення обробки операцій. Забезпечення високої доступності реалізується шляхом використання операторів Kubernetes (наприклад, Patroni для PostgreSQL або MongoDB Operator), механізмів автоматичного відновлення (self—healing), розподілених транзакцій та балансування навантаження. Таким чином система отримує здатність продовжувати роботу навіть за умов відмови окремих вузлів або цілих сегментів інфраструктури, що є фундаментальним для проєктування відмовостійких сервісів із апаратними ключами.

Отже, архітектура відмовостійких сервісів із використанням апаратних ключів має поєднувати кілька взаємопов’язаних рівнів — фізичний, програмний та оркестраційний. Ефективність таких систем визначається не лише наявністю резервних вузлів чи дублюванням обладнання, а й узгодженою роботою механізмів моніторингу, балансування навантаження, серіалізації доступу до ключа та контролю безпеки. Архітекрута Active—Active (рис 1.3) повністю відповідає вимогам до доступності та оптимізує використання обчислювальних ресурсів (серверів). Кількість серверів можна збільшувати відповідно до навантаження що забезпечить високу доступність. Отже для подови програмного продукту буде доцільно обрати саме її.

1.4 Визначення вимог до відмовостійкості систем

Відмовостійкість — це здатність системи зберігати свою функціональність у разі виникнення збоїв окремих компонентів, забезпечуючи безперервність виконання основних процесів. Для серверів ліцензування, які взаємодіють з апаратними ключами, вимоги до відмовостійкості є критично важливими, оскільки будь—яка втрата доступу до ключа призводить до зупинки роботи програмного забезпечення, що може спричинити значні фінансові та операційні втрати [16].

Загальні вимоги до відмовостійких систем визначаються на основі трьох ключових критеріїв: доступності (availability), надійності (reliability) та відновлюваності (recoverability) (рис. 1.4).

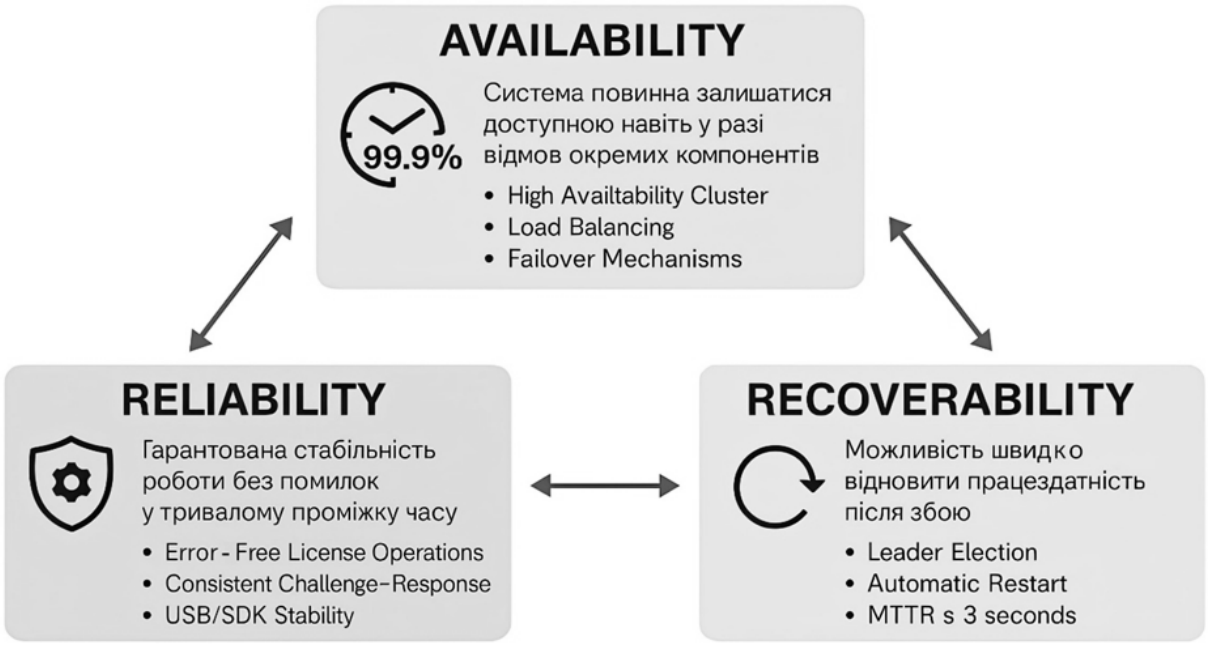


Рисунок 1.4 — Основні складові відмовостійкості системи ліцензування з апаратними ключами: доступність, надійність та відновлюваність

Доступність характеризує відсоток часу, протягом якого система працює безперебійно. Для критичних сервісів, таких як ліцензійні сервери, типовими орієнтирами є показники 99.99% (так звана “чотири дев’ятки”) або вище. Це означає, що сумарний час простою не повинен перевищувати кількох хвилин на місяць на таблиці 1.1. [17]

Таблиця 1.1 — Рівні доступності сервісів

Відсоток безвідмовної роботи	Щомісячний простій	Щорічний простій
99.5%	3,6 години	43,8 години
99,9%	43,8 хвилини	8,76 години
99,99%	4,38 хвилини	52,56 хвилини
99,999%	26,3 секунди	5,26 хвилини

Надійність визначається здатністю системи виконувати свої функції без збоїв протягом певного періоду часу, а відновлюваність — часом, необхідним для повернення системи до робочого стану після відмови.

Для забезпечення таких характеристик у системах із апаратними ключами мають бути реалізовані певні технічні й організаційні вимоги.

По—перше, необхідно передбачити фізичне резервування компонентів — використання декількох вузлів або серверів, які можуть дублювати функції основного у разі його виходу з ладу. Якщо ключ є єдиним фізичним пристроєм, то слід передбачити наявність резервного ключа з ідентичними правами або використання схеми “primary—secondary” із можливістю швидкого перемикання доступу.

По—друге, система повинна мати механізми автоматичного виявлення відмов і перемикання (failover). Це передбачає безперервний моніторинг стану апаратного ключа, ліцензійного демона, мережевого з’єднання та контейнерів, у яких вони працюють. При втраті зв’язку або виявленні помилки система має самостійно ініціювати перезапуск або переключення на резервний вузол без втручання оператора.

По—третє, важливим є забезпечення цілісності даних та синхронізації станів між активними й резервними компонентами. Усі зміни ліцензійних даних, лічильників або журналів доступу повинні зберігатися у транзакційному вигляді та передаватися в резервну копію для уникнення втрати інформації при відмові.

Ще однією вимогою є ізоляція відмов, тобто запобігання каскадному впливу збоїв. Вихід з ладу одного вузла або контейнера не повинен впливати на

стабільність усього кластера. Для цього використовуються механізми оркестрації, такі як PodDisruptionBudget, anti—affinity, node taints та topology spread constraints, які обмежують кількість одночасно недоступних компонентів і забезпечують їх розподіл по різних фізичних або віртуальних хостах.

Важливо також визначити вимоги до моніторингу та спостережуваності. Система має збирати показники часу відгуку, кількості успішних та неуспішних запитів, стану ключа, продуктивності мережі, використання ресурсів та інших параметрів, що характеризують її стабільність. Наявність таких метрик дозволяє проактивно виявляти проблеми та вживати заходів до настання критичних відмов.

Не менш значущим є питання безпеки, яке тісно пов’язане з відмовостійкістю. Будь—яке втручання у роботу апаратного ключа, спроби несанкціонованого підключення або підміни пристрою можуть призвести до зупинки системи. Тому важливо реалізувати контроль автентичності компонентів (mTLS), керування

Щодо експлуатаційних вимог, система повинна підтримувати оновлення без простоїв (rolling updates) і мати механізми rollback у разі невдалого оновлення програмного забезпечення. Це особливо актуально для середовищ, у яких обслуговування ліцензійного сервера відбувається без зупинки критичних бізнес—процесів.

Крім того, важливо забезпечити географічну відмовостійкість — можливість роботи системи у разі втрати цілого дата—центру або мережевого сегмента. Для цього застосовується реплікація ключів або ліцензійних серверів у різних зонах доступності (Availability Zones), синхронізація станів та глобальне балансування навантаження з автоматичним маршрутизацією трафіку [18].

Також однією з ключових вимог до відмовостійкості апаратного ключа є його здатність продовжувати роботу у випадку часткової втрати доступу до окремих компонентів або інфраструктурних сервісів. Оскільки апаратний ключ є фізичним носієм логіки ліцензування, його відмова безпосередньо впливає на доступність програмного забезпечення. Тому пристрій має підтримувати

автономний режим функціонування, зберігати працездатність при тимчасовій втраті мережевого з’єднання, забезпечувати захищену обробку запитів та гарантувати цілісність криптографічних операцій незалежно від стану зовнішніх ресурсів. До базових вимог входять також захист від фізичного доступу, стійкість до перебоїв живлення та здатність до автоматичного відновлення після перезапуску.

Окремим аспектом відмовостійкості апаратного ключа є наявність GPS—модуля, який забезпечує контроль місцезнаходження пристрою та виступає додатковим механізмом виявлення аномалій. У випадку втрати супутникового сигналу, появи надмірних похибок координат або повної недоступності GPS, ключ повинен переходити в безпечний режим роботи, у якому криптографічні операції виконуються з використанням резервних параметрів або обмеженої функціональності. Це запобігає ситуаціям, коли тимчасові порушення у прийомі GPS—сигналу можуть призвести до збою ліцензійної системи або помилкового блокування користувачів. Важливою вимогою є також наявність допустимого діапазону похибки, який дозволяє уникнути помилкових спрацювань захисту під час природних флуктуацій супутникових даних.

GPS—модуль виконує також функцію сенсора безпеки, що дозволяє виявляти спроби фізичного переміщення або викрадення ключа. Відповідно, вимоги до відмовостійкості включають механізм поведінки пристрою у разі різкої зміни координат або виходу за межі дозволеного геоперіметру. У таких сценаріях апаратний ключ повинен негайно активувати захисні процедури: перейти у режим часткового або повного блокування, змінити параметри шифрування, сформувати подію безпеки для серверної інфраструктури та передати відповідні сповіщення до системи моніторингу. Це забезпечує можливість оперативного реагування та мінімізує ризик компрометації, навіть якщо фізичний доступ до пристрою був отриманий зловмисником.

Отже, узагальнюючи, основними вимогами до відмовостійких систем ліцензування з апаратними ключами є:

—наявність резервування фізичних і програмних компонентів;

—автоматичне виявлення відмов і перемикання на резерв;

—забезпечення цілісності й узгодженості даних;

—ізоляція збоїв та обмеження їхнього впливу;

—безперервний моніторинг, аудит і звітність;

—гарантії безпеки доступу до ключів;

—підтримка оновлень без зупинки роботи;

—можливість географічного відновлення після катастрофічних відмов.

Виконання зазначених вимог є передумовою побудови надійної системи ліцензування, здатної функціонувати в умовах сучасних контейнеризованих середовищ, забезпечуючи високу доступність, мінімальний час простою та стабільну роботу критичних сервісів.

Отже, у першому розділі було проведено аналіз сучасних систем ліцензування з використанням апаратних ключів, визначено їхнє місце у сучасних програмних продуктах та досліджено архітектурні підходи до підвищення їхньої відмовостійкості. Розглянуто роль ліцензування як механізму захисту авторських прав і як технічного інструмента, що забезпечує контроль за використанням програмного забезпечення.

Проведений аналіз показав, що апаратні ключі залишаються одним із найбільш надійних засобів захисту від несанкціонованого доступу, проте їх використання ускладнює масштабування й автоматизацію у хмарних і контейнеризованих середовищах. Основні переваги таких систем — високий рівень безпеки, неможливість підробки та централізований контроль ліцензій. Серед недоліків — залежність від фізичного пристрою, складність резервування та інтеграції в сучасні інфраструктури.

Розглянуті архітектурні підходи, такі як моделі active—passive та active—active, використання шлюзів доступу, механізмів оркестрації контейнерів (Kubernetes Device Plugin, PodDisruptionBudget, leader election тощо), показали, що відмовостійкість може бути досягнута шляхом поєднання програмних і апаратних рішень. Визначено ключові вимоги до таких систем: автоматичне

відновлення після збоїв, ізоляція відмов, безперервний моніторинг, безпечне управління ключами та підтримка оновлень без простоїв.

Таким чином, у результаті проведеного дослідження було сформульовано основні принципи побудови надійних та безпечних систем ліцензування, що стануть теоретичною і практичною основою для подальшої розробки методу забезпечення відмовостійкості серверів ліцензування в контейнеризованому середовищі.

2 АПАРАТНА ЧАСТИНА МЕТОДУ ЗАБЕЗПЕЧЕННЯ ВІДМОВОСТІЙКОСТІ ТА ДОСТУПНОСТІ СЕРВЕРІВ ЛІЦЕНЗУВАННЯ

2.1 Метод забезпечення відмовостійкості та доступності серверів ліцензування з апаратними ключами в контейнеризованому середовищі

Розроблення методу забезпечення відмовостійкості серверів ліцензування ґрунтується на необхідності адаптації традиційних підходів до роботи з апаратними ключами в умовах сучасних контейнеризованих і хмарних середовищ. Головна проблема полягає у тому, що фізичний апаратний ключ, будучи матеріальним носієм ліцензії, не може бути безпосередньо масштабований або дубльований як програмний ресурс. Тому запропонований метод має забезпечити логічне розширення доступності такого ключа за рахунок розподілу обчислювальних навантажень, організації керованого доступу до нього та використання механізмів автоматичного відновлення у разі збоїв.

Сутність розробленого методу полягає у поєднанні програмно—апаратного резервування із засобами оркестрації контейнерів. Основна ідея — відокремити апаратний ресурс (ключ) від програмних компонентів, які його використовують, і надати до нього доступ через спеціалізований сервіс—шлюз. Цей сервіс працює як єдина точка взаємодії між контейнерами, що потребують ліцензії, та фізичним ключем, забезпечуючи контроль черги запитів, автентифікацію клієнтів, перевірку стану пристрою та автоматичне перемикання у разі збою.

На рис. 2.1 розглянуто реалізацію запропонованого у МКР методу на базі використання трирівневої архітектури.

Перший рівень — фізичний, до складу якого входять вузли з під’єднаними апаратними ключами. На цьому рівні реалізуються функції апаратного моніторингу, виявлення несправностей та відновлення роботи USB—портів. Для цього використовується агент спос стереження, який контролює стан ключів, виконує періодичне опитування, фіксує підключення та відключення, а також передає метрики у систему моніторингу.

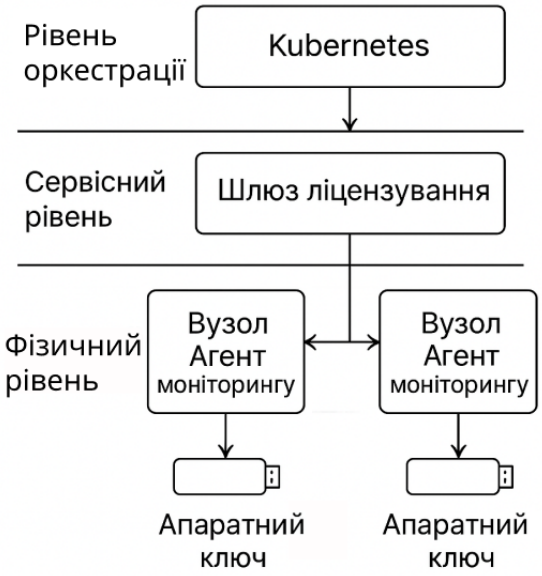


Рисунок 2.1 — Методу доступу до апаратного ключа на базі трирівневої архітектури

Другий рівень — сервісний, який реалізує програмну логіку взаємодії контейнерів з апаратним ключем. У межах цього рівня створюється окремий контейнер—шлюз (License Gateway), який працює у режимі active—active або active—passive. Шлюз взаємодіє з фізичним ключем за допомогою драйвера або SDK від постачальника (наприклад, CodeMeter, Sentinel або FlexNet) і надає інтерфейс REST/gRPC для інших мікросервісів. У разі відмови основного вузла система автоматично обирає новий активний програмий вузол за допомогою механізму leader election, що мінімізує час простою.

Третій рівень — оркестраційний, який забезпечує керування контейнерами, моніторинг стану системи, автоматичне масштабування та балансування навантаження. Для реалізації цього рівня використовується ПЗ Kubernetes, що дозволяє розподіляти сервіси між вузлами, проводити оновлення без простоїв (rolling updates), а також реалізовувати політики доступності за допомогою таких інструментів, як PodDisruptionBudget, node affinity/anti—affinity та topology spread constraints.

Ключовим елементом методу є впровадження механізму динамічного виявлення та перепідключення ключів, який у разі фізичного від’єднання прист-

рою або втрати зв’язку ініціює повторне підключення без потреби зупинки контейнерів. Це досягається завдяки спеціальному драйверу—посереднику, який взаємодіє із системними службами операційної системи та перехоплює події USB.

Ще одним важливим компонентом є система моніторингу та виявлення помилок у системі, що інтегрується з Prometheus або Grafana. Вона відстежує метрики доступності ключа, затримку запитів, кількість активних ліцензій, частоту збоїв і тривалість їхнього усунення. Зібрані дані дозволяють оцінювати фактичний рівень доступності сервісу (SLA, SLO) та своєчасно реагувати на потенційні відмови [19].

У межах розробленого методу також реалізується механізм резервного доступу до апаратного ключа. Для цього створюються два вузли — основний і резервний, між якими налаштовано реплікацію метаданих ліцензій і синхронізацію станів. У разі відмови основного вузла система автоматично переведе навантаження на резервний за допомогою протоколів Keepalived або VRRP, що дозволяє зберегти безперервність обслуговування клієнтів.

Для підвищення безпеки метод передбачає використання взаємної автентифікації (mTLS) між контейнерами та шлюзом, контролю доступу (RBAC) на рівні кластеру та керування секретами через системи Vault або KMS. Таким чином забезпечується захист як переданих даних, так і апаратних ресурсів від несанкціонованого доступу (рис. 2.2).

Запропонований метод поєднує переваги апаратного захисту з можливостями контейнерної оркестрації, що дозволяє досягти високого рівня відмовостійкості, гнучкості й масштабованості систем ліцензування. Його впровадження дає змогу зменшити час простою ліцензійних серверів, мінімізувати ризики втрати доступу до апаратних ключів і підвищити ефективність управління ліцензійною інфраструктурою у сучасних корпоративних середовищах.

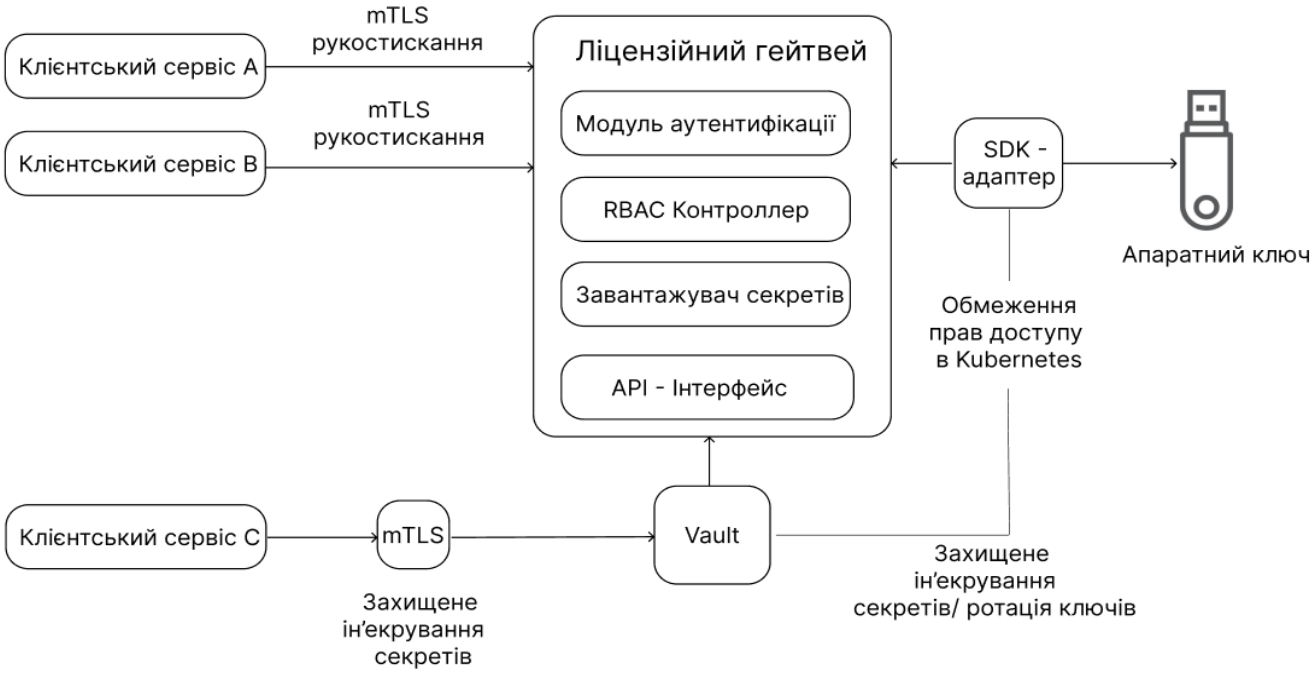


Рисунок 2.2 — Запропонована схема захищеної комунікації між контейнерами та сервісом—шлюзом

2.2 Апаратна частина апаратного ключа. Структурна модель

Проєктування апаратної частини апаратного ключа є одним із ключових етапів створення надійної системи ліцензування, оскільки саме на цьому рівні формується фізичний носій криптографічної інформації та механізм її безпечної обробки. Апаратний ключ виконує функції зберігання секретних параметрів, виконання криптографічних операцій і забезпечення автентичності, виступаючи центральною ланкою між програмним забезпеченням та захищеною ліцензійною логікою.

У межах цієї роботи апаратною основою ключа обрано мікрокомп'ютер Raspberry Pi 5 (рис. 2.3), який поєднує високу обчислювальну продуктивність, доступність та широкий набір інтерфейсів, необхідних для реалізації кастомного USB—пристрою.

Пристрій оснащений портами USB та Ethernet, що розширює можливості його інтеграції в інфраструктуру. Порт USB використовується для підключення GPS—модуля, який забезпечує визначення координат у процесі роботи, тоді як

інтерфейс Ethernet слугує для підключення апаратного ключа до серверної мережі. Це дозволяє програмному забезпеченню, встановленому на мікроконтролері, отримувати власну IP—адресу та забезпечувати стабільну взаємодію з іншими компонентами системи.

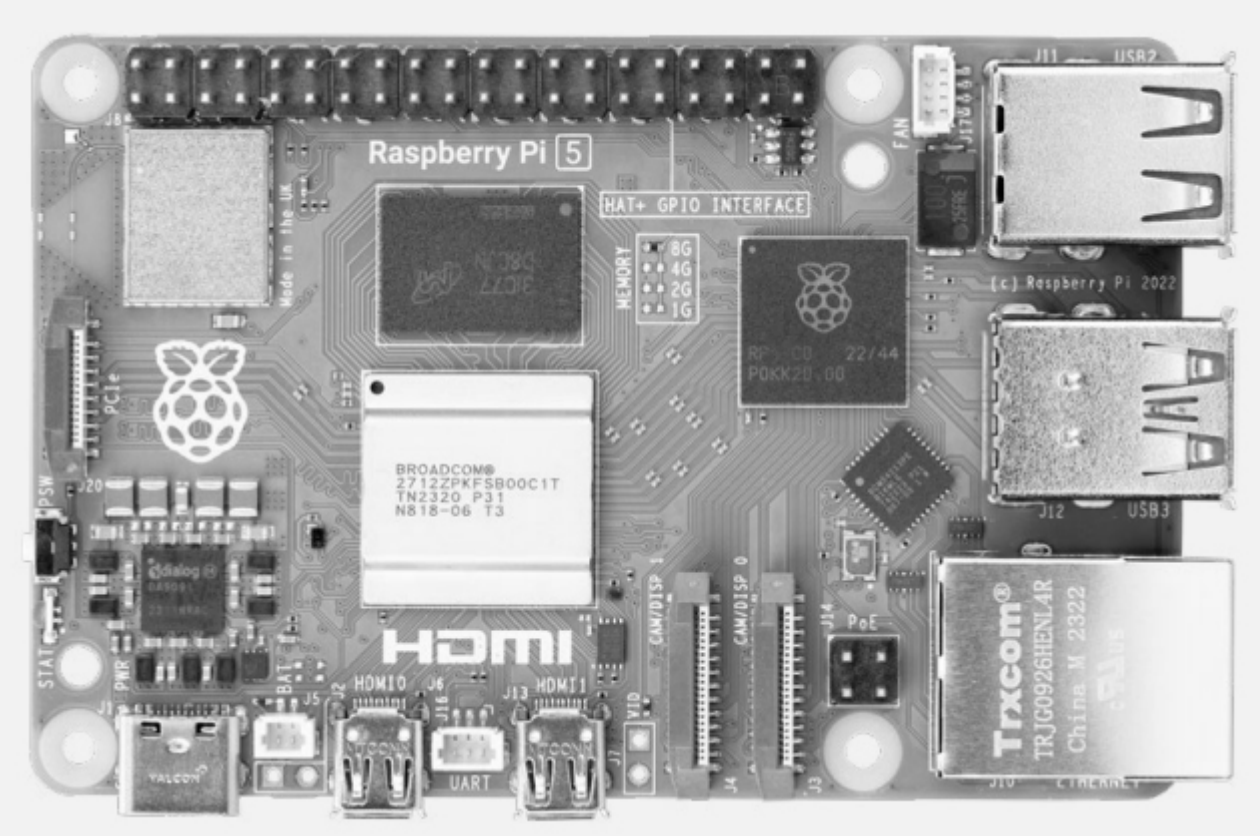


Рисунок 2.3 — Мікрокомп’ютер Raspberry Pi 5

Для розміщення апаратного ключа на основі Raspberry Pi 5 із підключеним GPS—модулем NEO—8M та інтерфейсом Ethernet використовується спеціалізований промисловий корпус GEEKOM IT 13 Mini PC (рис 2.4).

Такий корпус забезпечує захист пристрою від механічних впливів, електромагнітних завад та несанкціонованого фізичного доступу, що є критично важливим у контексті розробки засобів апаратної безпеки. Алюмінієві моделі корпусів також виконують функцію пасивного радіатора, відводячи тепло від високопродуктивного процесора Raspberry Pi 5 та забезпечуючи стабільність його роботи під час виконання криптографічних операцій.

Рисунок 2.4 — Промисловий корпус GEEKOM IT 13 Mini PC для оболонки апаратного ключа

Корпус повинен мати достатній внутрішній простір для розміщення самої плати Raspberry Pi 5, GPS—модуля NEO—8M та необхідних комунікаційних інтерфейсів. Наявність зовнішніх отворів під роз’єми Ethernet і USB забезпечує можливість прямої інтеграції апаратного ключа до серверної інфраструктури та підключення GPS—антени. З огляду на те, що робота GPS—модуля потребує якісного прийому супутникового сигналу, конструкція корпусу передбачає використання зовнішньої SMA—антени або встановлення внутрішньої керамічної антени у разі застосування пластикового корпусу з низьким рівнем екранування.

Розміри наступні: ширина — 12см, довжина — 11см та висота — 5см, та вага 0,652 кг.

Особливу увагу приділено теплотехнічним та інженерним характеристикам корпусу. Raspberry Pi 5 має високий тепловий пакет, тому корпус має забезпечувати можливість пасивного або комбінованого (пасивно—активного) охолодження. Алюмінієві промислові корпуси, такі як GEEKOM IT 13 Mini PC,

дозволяють знизити температуру процесора на 10—20 °C, підвищуючи надійність роботи пристрою. У комплексі такі корпуси забезпечують оптимальні умови для стабільної роботи Ethernet—інтерфейсу, GPS—модуля та криптографічних алгоритмів, зберігаючи при цьому компактність і захищеність апаратного ключа в умовах реальної експлуатації [20].

Для підвищення рівня безпеки до структури апаратного ключа інтегрується GPS—модуль (рис. 2.5), який забезпечує постійний контроль фактичного місцезнаходження пристрою. Як навігаційний компонент використано модуль NEO—8M, що характеризується високою точністю визначення координат та швидким отриманням сигналу від супутникових систем.

Рисунок 2.5 — GPS модуль NEO—8M

Застосування GPS—модуля дає змогу реалізувати механізми географічного прив’язування криптографічних операцій, виявляти несанкціоноване переміщення ключа та автоматично активувати захисні режими у разі виходу за визначений геоперіметр, що суттєво підвищує стійкість системи до фізичних атак. Хоча точність визначення координат модуля становить до двох метрів, з практичних міркувань програмної реалізації доцільно збільшити припустиму похибку до п’ятдесяти метрів. Такий підхід запобігає помилковим спрацюванням захисних механізмів у разі тимчасових збоїв супутникового сигналу та гарантує безперебійну роботу програмного забезпечення навіть за умов нестабільного GPS—покриття.

Структурна модель апаратного ключа (рис. 2.6) являє собою апаратно—програмний комплекс, що поєднує кілька функціональних модулів, інтегрованих у єдиний пристрій. Основою конструкції є міні—комп’ютер GEEKOM IT13 Mini PC, корпус якого забезпечує достатній внутрішній простір для розміщення плати Raspberry Pi 5, GPS—модуля NEO—8M, додаткових сенсорів та допоміжних комунікаційних інтерфейсів. Завдяки компактності та продуманій конструкції корпус дозволяє розмістити апаратні компоненти таким чином, щоб забезпечити захист від фізичного доступу, стабільне охолодження та можливість виведення зовнішніх інтерфейсів.

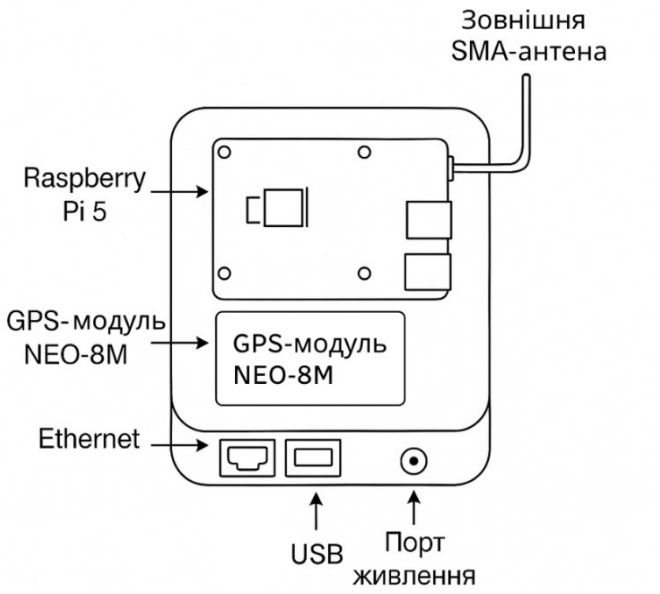


Рисунок 2.6 — Структурна модель апаратного ключа

Розглянемо будову окреми модулів апаратного ключа.

У конструкцію апаратного ключа входить міні—комп’ютер Raspberry Pi 5, який виконує основну обчислювальну логіку: обробку криптографічних операцій, виконання ліцензійних алгоритмів, взаємодію з сервером через Ethernet та керування периферійними модулями. Плата під’єднується до внутрішнього живлення корпусу та має прямий доступ до мережевого інтерфейсу, що дозволяє мікрокомп’ютеру отримувати статичну або динамічну IP—адресу та працювати в клієнтсько—серверній інфраструктурі.

Другим ключовим елементом моделі є GPS—модуль NEO—8M, який відповідає за визначення географічних координат і контроль місцезнаходження апаратного ключа. Він з'єднується з Raspberry Pi 5 через USB або UART та може використовувати як внутрішню керамічну антену, так і зовнішню SMA—антену, використання якої забезпечує стабільний прийом сигналу всередині приміщень або в умовах значного екранування. GPS—модуль виконує функції захисту, включно з визначенням аномального переміщення, ініціюванням захисного режиму при зміні координат та виявленням втрати супутникового сигналу.

Корпус GEEKOM IT13 Mini PC використовується як зовнішній захисний контейнер, який поєднує компактність, міцність і достатній простір для встановлення обладнання. Він оснащений вентиляційними прорізами, монтажними отворами та можливістю фіксації додаткових компонентів. На корпус виводяться ключові інтерфейси:

– Ethernet — для підключення апаратного ключа до серверної мережі;

– USB — для інтеграції GPS—модуля та можливого обслуговування пристрою;

– SMA—роз'єм — для зовнішньої GPS—антени;

– порт живлення — для стабільної роботи усіх внутрішніх модулів.

Структурна модель також передбачає наявність внутрішньої схеми моніторингу, яка забезпечує контроль температури, живлення, працездатності модулів та стану GPS—сигналу. Усі події передаються до серверної інфраструктури, що дозволяє системі ліцензування реагувати на будь—які відхилення у роботі апаратного ключа.

Захист апаратної частини реалізується шляхом використання комплексу інженерно—технічних та програмно—криптографічних заходів. Корпус апаратного ключа виготовляється з антивандальних матеріалів, а кристал мікросхеми додатково герметизується компаундом, що унеможливлює доступ до внутрішніх елементів та суттєво ускладнює фізичний реверс—інжиніринг.

Окрім фізичного захисту, критично важливою є безпека прошивки мікроконтролера. Для цього застосовуються такі механізми:

— блокування інтерфейсів відлагодження (SWD/JTAG) після встановлення прошивки, що унеможливлює її пряме зчитування;

— шифрування прошивки перед записом у пам’ять мікроконтролера, що робить недоступним її аналіз навіть у разі фізичного доступу до пам'яті;

— використання контрольних сум та цифрових підписів, які забезпечують верифікацію цілісності прошивки під час запуску.

Для запобігання витоку критичних даних і ключової логіки, у мікросхемі реалізовано механізм самознищення інформації при спробі несанкціонованого доступу. Такий механізм активується у випадках:

—виявлення доступу через заблокований SWD/JTAG інтерфейс;

—спроби зчитування пам’яті за допомогою сторонніх програматорів;

—аномальної поведінки живлення або тактової частоти, що може свідчити про використання атак типу fault—injection;

—багаторазового введення некоректного адміністративного пароля.

У разі активації механізму захисту мікроконтролер здійснює безповоротне очищення пам’яті, включно з криптографічними ключами, частиною логіки обчислень та службовими даними. Це робить неможливим відтворення алгоритмів, що зберігалися на апаратному ключі, та повністю нівелює спроби реверс—інжинірингу або фізичного копіювання пристрою.

Використання таких багаторівневих заходів захисту забезпечує високий рівень стійкості апаратного ключа до фізичного злому, аналізу прошивки та крадіжки інтелектуальної власності.

Для запобігання фізичній підміні апаратного ключа застосовуються програмні механізми захисту. У випадку, коли ключ виконує лише бінарне повернення результату (true/false), що інтерпретується як авторизований або неавторизований стан, його функціональність може бути відносно легко емульована сторонніми засобами. Натомість розміщення на апаратному ключі критично важливих обчислювальних процедур, необхідних для коректного функціонування програмного забезпечення, робить підміну фактично неможливою. Відтворення таких алгоритмів вимагало б повного дублювання

апаратної логіки, що є технічно вкрай складним та економічно недоцільним. У разі спроби фізичного втручання працює tamper—система, яка блокує пристрій або стирає секретні дані. Контролер може виконувати самотестування, перевірку CRC, контроль напруги та температури, що гарантує стабільну роботу навіть в умовах коливань живлення або температурних перепадів.

Окремим типом загроз є фізичне проникнення зловмисника до дата—центру з метою викрадення апаратного ключа або флеш—модуля, що містить критично важливі дані та логіку шифрування. У такому випадку система захисту переходить у режим підвищеної безпеки та активує низку захисних механізмів:

По—перше, апаратний ключ оснащений вбудованим GPS—модулем який постійно контролює своє місцезнаходження. У разі зміни координат, що не відповідає дозволеним зонам експлуатації (геоперіметру), або раптового зникнення живлення у захищеному середовищі, пристрій автоматично переходить у режим порушення безпеки. У цьому стані ключ змінює внутрішні параметри роботи та формує альтернативну криптографічну послідовність, відмінну від тієї, що використовувалася у штатному режимі. Внаслідок цього всі подальші операції шифрування та дешифрування стають некоректними для легітимної системи, що унеможливлює відновлення первинної логіки обчислень навіть при фізичному доступі до пристрою.

По—друге, програмна інфраструктура, що взаємодіє з ключем, здійснює постійний моніторинг його стану. При втраті зв’язку з датчиком місцезнаходження або при фіксації аномальних координат система генерує аварійні сповіщення (алерти) для адміністраторів. Такі сповіщення надходять у реальному часі через системи моніторингу та журналювання (наприклад, Prometheus Alertmanager або SIEM—платформи), що дозволяє оперативно виявити факт викрадення.

По—третє, у разі підтвердження викрадення апаратного ключа спеціальний модуль безпеки на сервері ініціює ревокацію криптографічних матеріалів, що були пов’язані з цим ключем. Це включає генерацію нових ключів

шифрування, перезапис конфігурації системи та блокування будь—яких операцій, пов’язаних із скомпрометованим пристроєм.

Таким чином, навіть у випадку фізичного доступу до дата—центру та викрадення флеш—модуля, зловмисник не отримує можливості зламати систему або відновити логіку її роботи. Завдяки поєднанню GPS—контролю, автоматичного переходу в захисний режим, механізмів алертингу та криптографічної ревокації забезпечується високий рівень стійкості до фізичних атак та захист від компрометації інтелектуальної власності.

У розробці особливу увагу приділено забезпеченню сумісності апаратного ключа з контейнеризованими середовищами. Апаратний ключ під’єднується до серверної інфраструктури через Ethernet—інтерфейс, отримуючи статичну або динамічну IP—адресу, що дозволяє використовувати його як повноцінний мережевий пристрій у кластері. Після отримання IP—адреси ключ стає доступним для сервісів Kubernetes через внутрішню мережеву взаємодію, що значно підвищує його гнучкість і зменшує залежність від фізичного розміщення.

Для інтеграції апаратного ключа у Kubernetes—кластер використовується механізм Device Plugin, який автоматично позначає вузли, що мають доступ до ключа, та направляє відповідні поди саме на ці вузли. Такий підхід забезпечує гарантований доступ сервісів до апаратного ключа незалежно від того, на якому фізичному обладнанні вони розгорнуті. Крім того, апаратний ключ може працювати як через локальний USB, так і через USB—over—IP, що дозволяє розміщувати пристрій окремо від вузлів кластера без втрати функціональності (рис. 2.7).

Для підвищення відмовостійкості інфраструктура доповнюється незалежними USB—хабами, резервованим живленням та можливістю гарячої заміни апаратного ключа без зупинки сервісів. Це забезпечує стабільну роботу системи навіть за умови часткових збоїв обладнання. Розроблена апаратна архітектура на базі Raspberry Pi 5 забезпечує високу гнучкість, захищеність та сумісність із сучасними обчислювальними середовищами, дозволяючи реалізувати всі необхідні функції ліцензійного апаратного ключа та відповідати вимогам

безпеки, автономності й надійності.

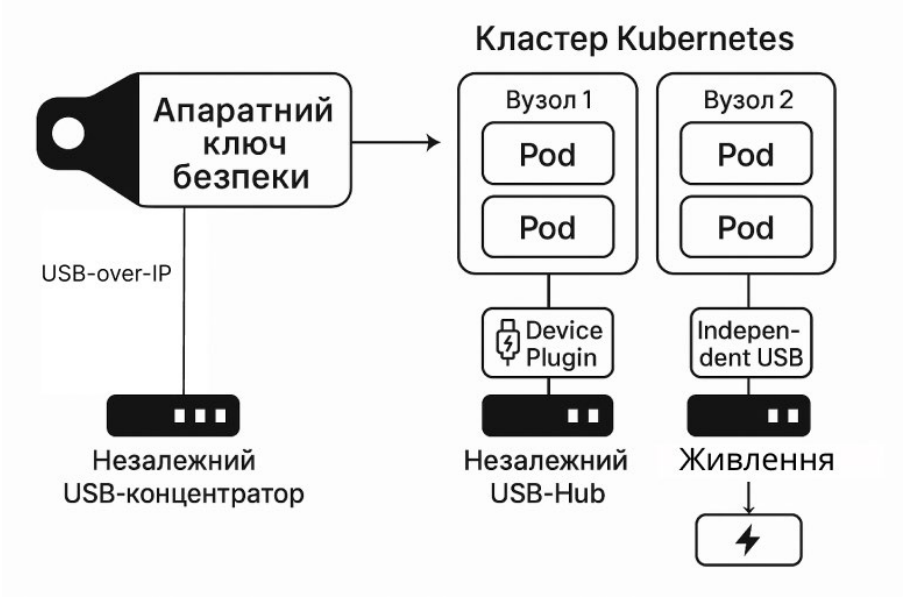


Рисунок 2.7 — Інтеграція апаратного ключа з Kubernetes—кластерами засобами USB / USB—over—IP та механізм Device—Plugin

2.3 Програмна модель доступу до апаратного ключа

Розглянемо програмну модель доступу до апаратного ключа, що представлено на рис. 2.8, яка покликана ізолювати фізичну специфіку апаратного ключа від прикладних сервісів та забезпечити керований, спостережуваний і відмовостійкий доступ до ліцензійного механізму.

Основною концепцією є впровадження проміжного шару сервісу—шлюзу ліцензування, який інкапсулює взаємодію з вендорським SDK, реалізує політики керування чергою запитів, механізми ідемпотентності операцій та аудит доступу, а також надає стандартизований інтерфейс взаємодії (REST/gRPC) для всіх споживачів у кластері. Такий підхід гарантує, що прикладні мікросервіси ніколи не звертаються до апаратного ключа напряму та залишаються незалежними від його фізичного розміщення, типу драйвера, особливостей встановлення або змін у вендорській реалізації [21].

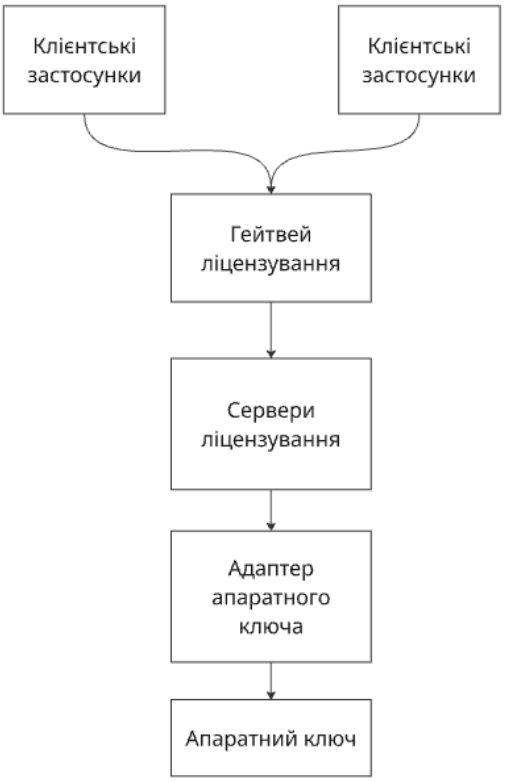


Рисунок 2.8 — Програмна модель доступу до апаратного ключа

Додатковою перевагою такого підходу є можливість забезпечення рівномірного розподілу навантаження між кількома апаратними ключами, у випадку якщо інфраструктура містить декілька вузлів, кожен з яких має власний фізичний носій ліцензії. Шлюз ліцензування може виконувати балансування запитів, автоматичне перемикання між ключами при відмовах, а також централізований моніторинг стану кожного пристрою. Це значно підвищує надійність роботи системи та мінімізує ризик простою сервісів у випадку збоїв на рівні окремого вузла.

Оскільки запропонований метод забезпечує максимально захищене, кероване та масштабоване використання апаратного ключа, клієнтам, що впроваджують подібну систему захисту свого програмного забезпечення, рекомендовано застосовувати саме таку архітектуру. Вона дозволяє досягти високої ефективності використання ключа, мінімізувати вплив на бізнес—логіку при збоях апаратної частини, а також забезпечує централізоване адміністрування, швидке розширення інфраструктури та відповідність вимогам інформаційної безпеки.

Логічна модель складається з трьох шарів (рис 2.9).

На периферії знаходиться шар клієнтських застосунків та мікросервісів бізнес—домену, що виконують запити на отримання або звільнення ліцензії в межах транзакції роботи користувача чи обчислювальної задачі. У центрі розташовано сервіс—шлюз, який приймає запити, автентифікує клієнтів за допомогою взаємних сертифікатів, проводить авторизацію за ролями та політиками, застосовує обмеження швидкості, формує чергу на доступ до ключа і гарантує коректне завершення життєвого циклу “checkout—use—return”. У нижньому шарі функціонує адаптер до апаратного ключа, що інкапсулює вендорське SDK, керує сесіями, виконує challenge—response та проводить локальну самодіагностику стану пристрою і драйверів.

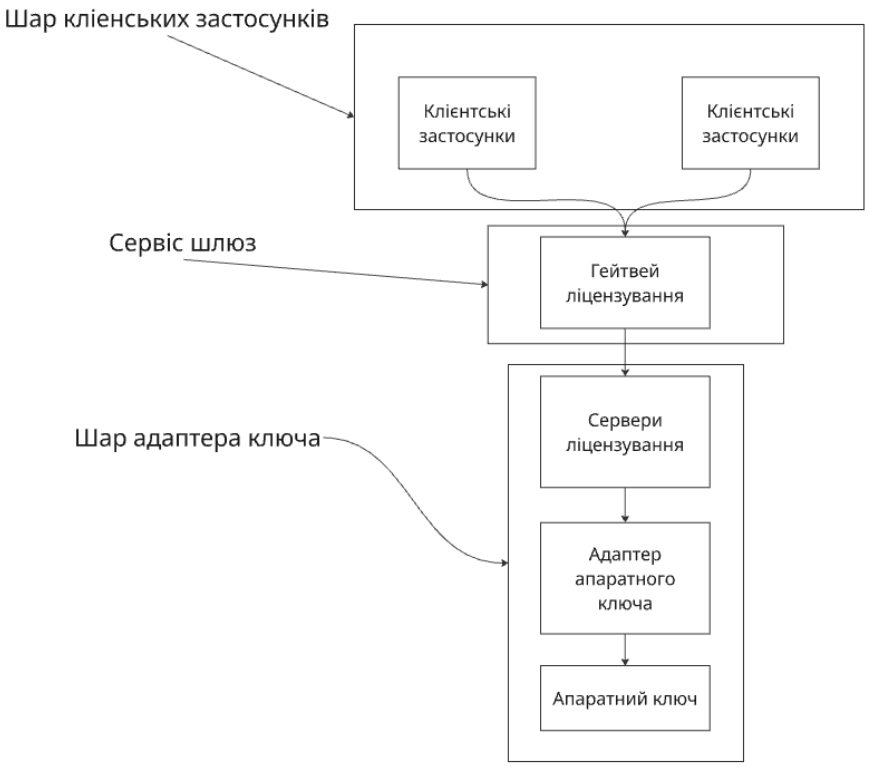


Рисунок 2.9 — Логічна модель доступу до апаратного ключа

Щоб забезпечити високу доступність, сервіс—шлюз розгортається у кількох репліках, між якими реалізовано механізм вибору лідера. Лідер має ексклюзивний доступ до адаптера ключа, тоді як інші інстанси обробляють автентифікацію, черги та метрики, але перенаправляють критичні виклики до

активного екземпляра через внутрішній RPC. Такий підхід дозволяє масштабувати фронт оброблення запитів і водночас уникати конкуренції за доступ до фізичного пристрою. Перемикання лідера відбувається автоматично за наявності ознак деградації (втрата heartbeat, збій SDK, зростання p95 часу `checkout()`), що мінімізує час простою.

Комунікація між компонентами будується на принципах нульової довіри. Усі виклики до шлюзу підписуються і шифруються (mTLS), сертифікати мають короткий строк дії та автоматично ротується через механізм видавця сертифікатів у кластері.

Авторизація здійснюється на основі атрибутів запиту: типу продукту, редакції, кількості одночасних місць, контексту проєкту та дозволених дій. Політики конфігуруються декларативно і зберігаються у централізованому сховищі конфігурацій, що дає змогу виконувати зміну правил без перезапуску сервісів і без впливу на працюючі сесії.

Стан системи розмежовується. Джерелом істини щодо кількості доступних ліцензій виступає сам апаратний ключ або вендорський демон; будь—які проміжні кеші слугують лише для телеметрії та оптимізації з'єднань. Для уникнення подвійних списань кожна операція `checkout()` має ідемпотентний ідентифікатор, а у разі повторного виклику в межах тайм—вікна шлюз повертає поточний стан без додаткового споживання ліцензії. У випадках втрати зв'язку реалізується механізм автоматичного відновлення сесії: при короткочасних збоях застосовується повтор із експоненційною затримкою, при триваліших — примусове повернення "завислих" оренд після контрольного тайм—ауту з подальшим аудитом [22].

Спостережуваність закладається як невід'ємна властивість архітектури. Кожен виклик фіксується у журналах із кореляційним ідентифікатором, метрики експортуються у систему моніторингу та включають рівень успішності, затримку на різних етапах обробки, глибину черги, кількість активних сесій, частоту відмов SDK і події апаратного рівня (detach/attach, помилки шини). На основі цих показників визначаються SLI та SLO, формується error budget і

налаштовуються попереджувальні сповіщення про вичерпання ліцензій, деградацію p95/p99 затримок і флапінг пристрою. Для розбору інцидентів передбачено знеособлений трейсинг запитів із мінімально необхідним набором полів.

З міркувань безпеки та керованості секрети (ключі шифрування, токени адміністрування, приватні сертифікати) зберігаються у зовнішньому сховищі секретів із журнальним доступом і політиками мінімальних привілеїв. Конфігурація під'єднання до вендорських сервісів, параметри тайм—аутів, обмеження на довжину сесії та профілі продуктивності постачаються через централізовані конфіг—джерела і підхоплюються під час виконання. Оновлення сервісів відбуваються за схемою без простоїв: для шлюзу застосовується стратегії поступового виведення з експлуатації, для адаптера — контрольована передача лідерства і від'єднання з коректним поверненням усіх активних оренд [23].

Особлива увага приділяється тестуванню стійкості. Архітектура включає сценарії хаос—експериментів: раптове від'єднання USB, штучні затримки на мережевому шляху, збої у вендорському SDK, падіння лідера та відмови вузла. Для кожного сценарію визначено очікувану поведінку (непорушність інваріантів, гарантоване повернення ліцензії, межі допустимої деградації затримок) і порядок відновлення. Функціональні тести перевіряють коректність бізнес—правил видачі, а навантажувальні оцінюють пікові `checkout rate` і стабільність під тривалим тиском з різними профілями запитів.

Взаємодія з прикладними командами спрощується через стабільний контракт API і офіційні SDK—обгортки, які інкапсулюють усі вимоги безпеки, генерацію ідемпотентних ключів, повтори з бюджетом і телеметрію. Таким чином, розробники бізнес—функцій отримують прозорий сервіс ліцензування як платформенну можливість, не занурюючись у деталі роботи з фізичним пристроєм. У підсумку архітектура забезпечує керованість, відмовостійкість і масштабованість, дозволяючи експлуатувати апаратний ключ як надійний

ресурс у контейнеризованому середовищі без компромісів щодо безпеки та доступності.

2.4 Метод підключення до апаратного ключа в кластерному середовищі

Проєктування методу доступу до апаратного ключа в кластерному середовищі є одним із ключових етапів розроблення підходу до забезпечення відмовостійкості, адже саме від нього залежить стабільність і безперервність роботи всієї системи ліцензування. Основною метою створення методу є розподіл запитів на використання апаратного ключа між кількома контейнерами або вузлами без порушення цілісності даних, уникнення одночасного доступу до одного ресурсу, а також забезпечення автоматичного відновлення зв’язку у випадку збоїв.

У кластерному середовищі, де програмні сервіси розподілені між різними вузлами, апаратний ключ залишається фізично унікальним пристроєм. Це створює певну асиметрію: програмна інфраструктура може масштабуватись майже необмежено, тоді як апаратний ресурс є обмеженим і має підтримувати лише один активний канал взаємодії в певний момент часу. Для вирішення цього протиріччя було розроблено підхід до централізованого доступу з механізмом серіалізації запитів, який реалізує логіку “один активний доступ — багато клієнтів”.

Процес уведення апаратного ключа в експлуатацію починається ще на етапі його фізичної доставки до датацентру й завершується повноцінною інтеграцією в кластер та системи моніторингу.

Розглянемо поетапне виконання методу підключення до апаратного ключа в кластерному середовищі (рис. 2.10).

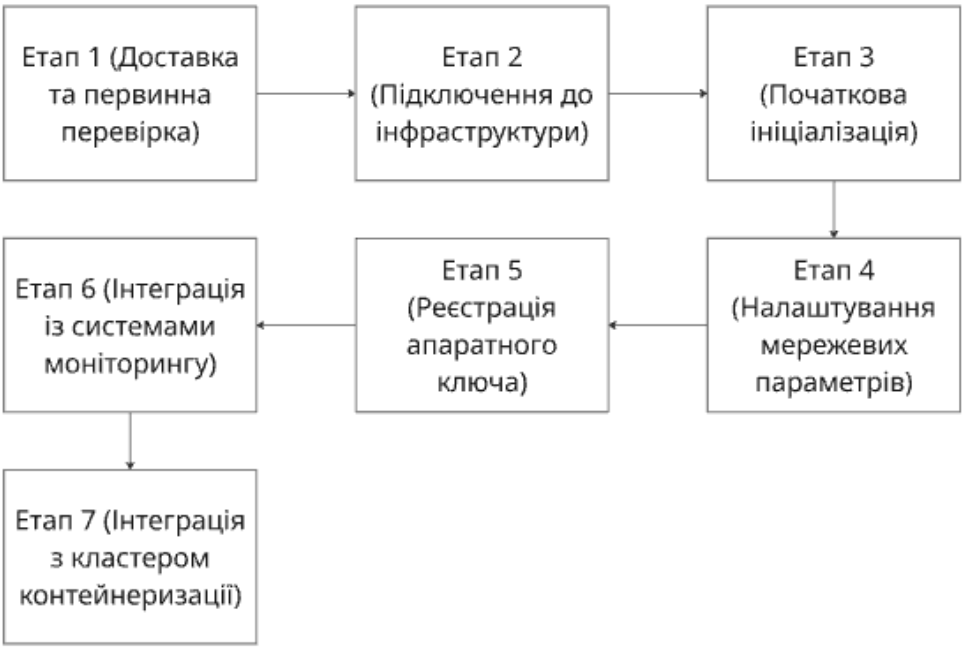


Рисунок 2.10 — Етапи методу підключення до апаратного ключа в кластерному середовищі

Етап 1 Доставка та первинна перевірка. Апаратний ключ доставляється до датацентру в опломбованій упаковці. Адміністратор розпаковує пристрій, виконує візуальну перевірку цілісності корпусу, пломб та відсутності механічних пошкоджень. За потреби результати перевірки фіксуються в журналі приймання обладнання.

Етап 2 Підключення до інфраструктури живлення та мережі. Пристрій під’єднується до джерела живлення згідно з технічною документацією. Після цього апаратний ключ підключається до мережевої інфраструктури датацентру через порт Ethernet відповідного серверного сегмента (наприклад, захищеної VLAN з доступом лише до сервісів ліцензування).

Етап 3 Початкова ініціалізація та введення адміністративного пароля. Після увімкнення живлення апаратний ключ переходить у початковий режим ініціалізації, у якому вимагає введення адміністративного пароля. Адміністратор за допомогою консольного доступу (SSH, серійний порт або веб—інтерфейс керування) задає або змінює початковий пароль на унікальний, що відповідає

політикам інформаційної безпеки організації. На цьому етапі також можуть створюватися додаткові облікові записи з обмеженими правами.

Етап 4 Налаштування мережевих параметрів та отримання IP—адреси. Далі виконується конфігурація мережі: пристрій може отримувати IP—адресу через DHCP або мати статичну адресу, зарезервовану в межах виділеного сегмента. Після присвоєння IP—адреси проводиться тестовий обмін пакетами (ping, перевірка доступу до службового порту) для підтвердження коректності мережевих налаштувань.

Етап 5 Реєстрація апаратного ключа в сервісі—шлюзі ліцензування. Отримана IP—адреса, ідентифікатор пристрою, серійний номер та інші атрибути вносяться в конфігурацію сервісу—шлюзу ліцензування. На цьому етапі ключ реєструється як окремий вузол у внутрішньому реєстрі ресурсів (наприклад, у конфігураційній базі або сховищі параметрів кластера), після чого шлюз отримує можливість здійснювати до нього контрольований доступ.

Етап 6 Інтеграція із системами моніторингу та виявлення помилок. Наступним кроком є підключення апаратного ключа до систем спостереження (моніторинг здоров’я сервісів, доступності, часу відповіді), а також налаштування правил сповіщень. Адміністратор конфігурує систему виявлення помилок на випадок втрати зв’язку з пристроєм, зміни його координат (згідно з GPS—модулем), повторних помилок автентифікації, некоректної роботи ліцензійного SDK тощо. Це забезпечує оперативне виявлення інцидентів безпеки та збоїв.

Етап 7 Інтеграція з кластером контейнеризації. На завершальному етапі апаратний ключ логічно “прив’язується” до кластерного середовища. У конфігурації Kubernetes або іншої системи оркестрації налаштовується сервіс ліцензування (gateway), який знає про IP—адресу ключа, політики доступу до нього, таймаути та параметри повторних спроб. Для вузлів, на яких розгортається сервіс—шлюз, можуть бути задані спеціальні мітки (labels), node affinity або tolerations, якщо ключ фізично прив’язаний до конкретного сегмента мережі або вузла.

На цьому етапі програмне забезпечення може повноцінно використовувати можливості апаратного ключа у своїй роботі. Для цього мікросервіси надсилають запити на IP—адресу, яку було встановлено під час початкової конфігурації пристрою. Оскільки взаємодія здійснюється в межах локальної мережі датацентру, затримки при обміні даними є мінімальними, що забезпечує високу швидкість обробки операцій ліцензування. Проте така конфігурація має суттєве обмеження: якщо програмний компонент, який взаємодіє з ключем, розгорнутий на одному фізичному сервері з пристроєм, то його відмова призведе до втрати доступності всієї системи ліцензування. Це суперечить вимогам до відмовостійкості, тому наступним етапом є розгортання повноцінного кластерного середовища.

Архітектурно метод доступу до апаратного ключа ґрунтується на концепції лідер—репліка, згідно з якою лише один вузол (лідер) у будь—який момент часу має ексклюзивний доступ до апаратного ключа (рис. 2.9). Усі інші вузли кластера перебувають у пасивному режимі та виконують роль реплік, готових перебрати функціональність лідера у разі його недоступності. Такий підхід розв’язує проблему єдиного фізичного ресурсу — апаратного ключа — забезпечуючи водночас масштабованість і високу доступність програмної інфраструктури [24].

Визначення активного вузла здійснюється за допомогою механізму leader election, що реалізується засобами Kubernetes через використання `ConfigMap`, `Lease API` або спеціалізованих контролерів. Після обрання лідера всі запити до апаратного ключа спрямовуються виключно до цього вузла через внутрішній сервіс—шлюз, що відповідає за серіалізацію запитів, контроль доступу та взаємодію з вендорським SDK. У разі збою лідера Kubernetes автоматично ініціює процедуру переобрання, і новий вузол миттєво перебирає роль активного. Це забезпечує безперервність роботи та мінімізує ризик простою ліцензійної системи навіть за умов часткових відмов інфраструктури.

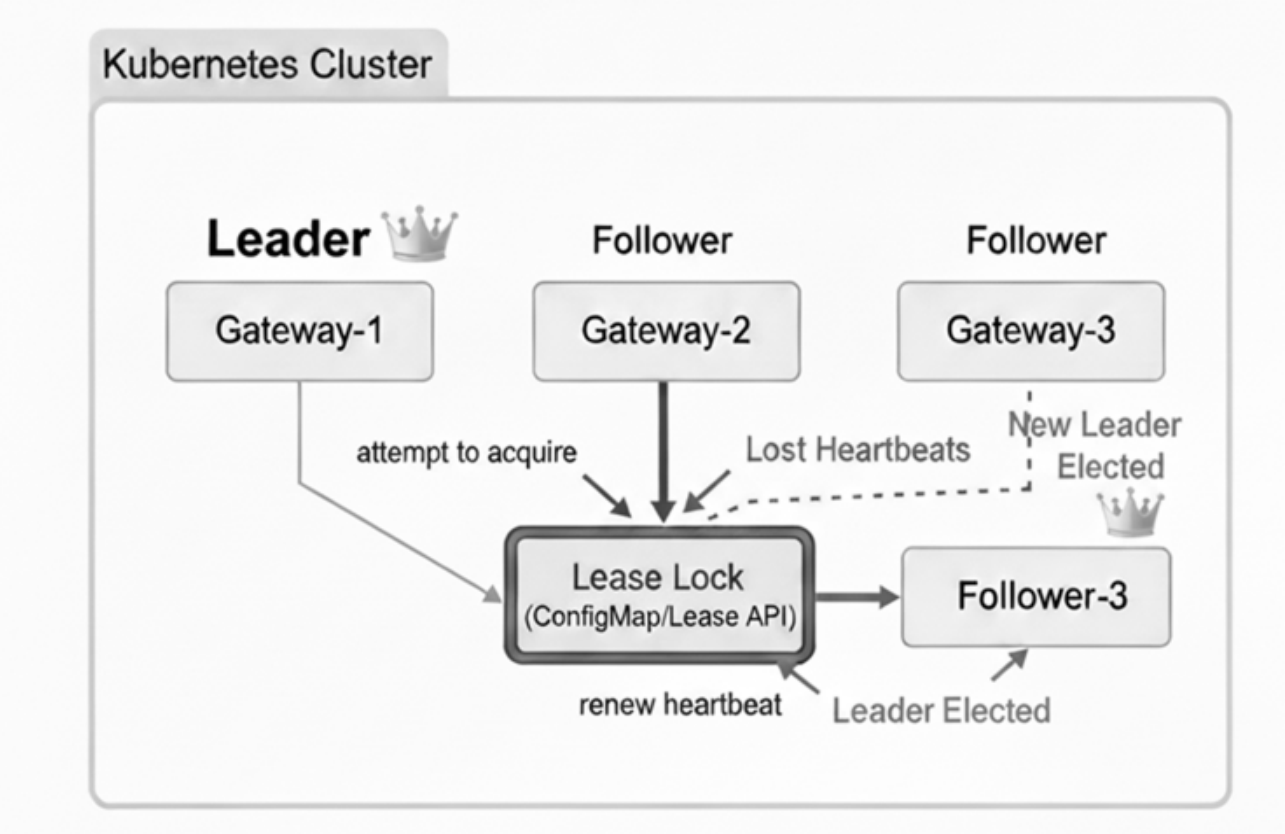


Рисунок 2.9 — Реалізація концепції лідер — репліка для доступу до апаратного ключа

Коли користувач або інший мікросервіс надсилає запит на отримання ліцензії, запит надходить у чергу шлюзу доступу до ключа. Кожному запиту присвоюється унікальний ідентифікатор та часовий штамп. Алгоритм перевіряє стан ключа, доступність лідера та відсутність активних транзакцій. Якщо ключ вільний, шлюз ініціює операцію `checkout()` через SDK виробника апаратного ключа, отримує токен доступу та передає його запитувачу. Токен має обмежений термін дії, після чого виконується автоматичне `return()` або подовження сесії при продовженні роботи користувача

Для уникнення колізій у черзі запитів застосовується стратегія ідемпотентності (рис. 2.10): повторні запити з тим самим ідентифікатором не породжують нових сесій. Це виключає дублювання ліцензій і гарантує, що навіть при мережевих затримках або повторних викликах клієнт отримає один і той

самий результат.

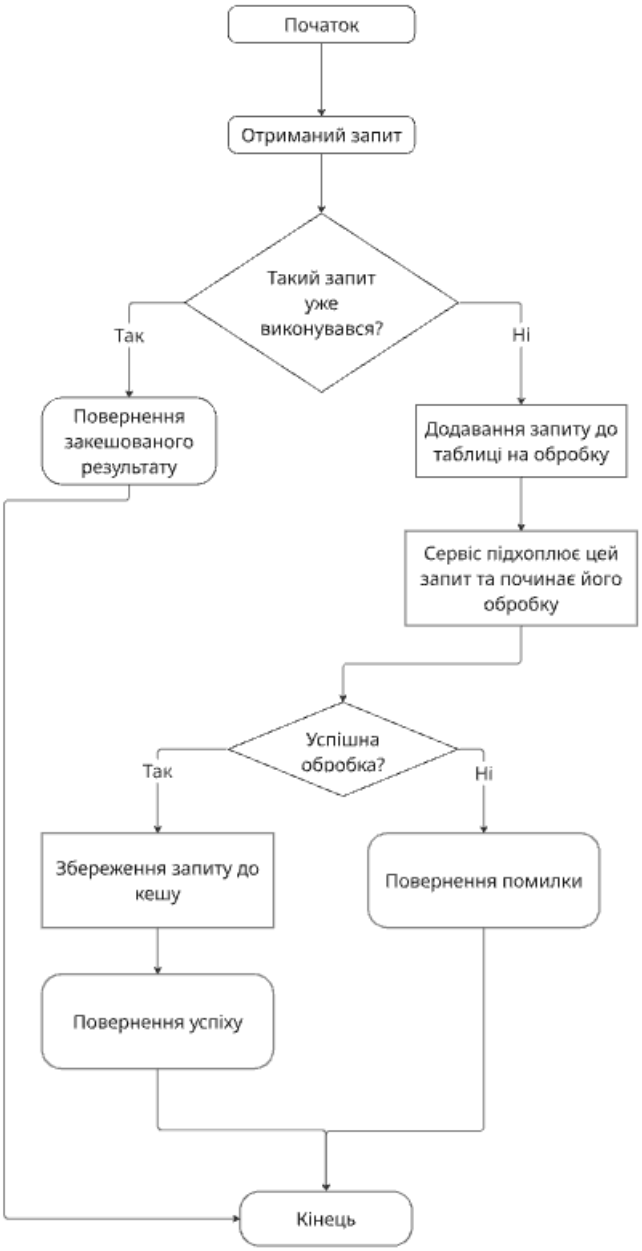


Рисунок 2.10 — Граф—схема досягнення ідемпотентності запитів

У разі виникнення збоїв передбачено кілька сценаріїв відновлення. Якщо втрачається з’єднання між ліцензійним шлюзом і апаратним ключем, активна сесія переходить у режим очікування з періодичними спробами перепідключення. Якщо впродовж певного тайм—ауту зв’язок не відновлюється, система ініціює процедуру failover: призначається новий лідер, а

попередній вузол від’єднується від ключа через механізм fencing, що унеможливлює одночасний доступ із двох джерел. Новий лідер відновлює роботу сервісу без втрати даних, виконуючи повторну автентифікацію та синхронізацію стану з попередніми сесіями. [25]

Додатково метод реалізує автоматичне балансування навантаження на рівні клієнтів (рис. 2.11).

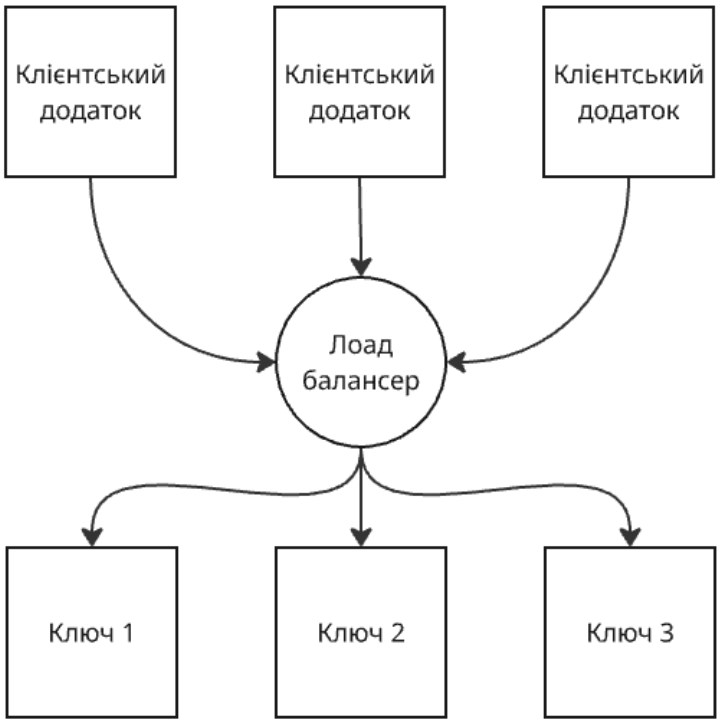


Рисунок 2.11 — Схема балансування навантаження на рівні клієнтів

Якщо система має кілька незалежних ключів або мережевих шлюзів, запити рівномірно розподіляються між ними з урахуванням поточної кількості активних сесій і часу відгуку. Це дозволяє оптимізувати використання ресурсів і уникнути перевантаження одного вузла.

Важливою складовою методу є захист від несанкціонованого доступу. Кожна транзакція супроводжується обов’язковою автентифікацією клієнта через протокол mTLS, а обмін даними з ключем виконується за зашифрованим каналом. Усі дії користувачів і системних процесів журналюються, що забезпечує повний аудит використання ліцензій і допомагає виявляти аномальну активність.

Таким чином, запропонований метод забезпечує:

—контрольований одночасний доступ до апаратного ключа без порушення цілісності даних;

—автоматичне відновлення після збоїв або втрати вузла—лідера;

—динамічне балансування навантаження в розподіленому середовищі;

—криптографічно захищену комунікацію;

—ідемпотентність операцій та аудит усіх транзакцій.

Поєднання цих характеристик дозволяє ефективно інтегрувати апаратний ключ у кластерну контейнеризовану архітектуру, забезпечуючи безперервність роботи системи ліцензування та підвищуючи її надійність і масштабованість.

Отже, у другому розділі було розроблено та обґрунтовано метод забезпечення відмовостійкості та доступності серверів ліцензування з апаратними ключами в контейнеризованому середовищі. Запропоновано трирівневу архітектуру, що включає фізичний рівень із резервованими вузлами, сервісний рівень зі шлюзом доступу до ключа та оркестраційний рівень, який забезпечує балансування, моніторинг і автоматичне відновлення системи.

Було спроєктовано апаратну частину ключа, яка поєднує в собі захист від фізичного втручання, криптографічні механізми перевірки автентичності та можливість роботи у контейнеризованих середовищах. Окрему увагу приділено інтеграції апаратного рівня з Kubernetes через Device Plugin, що забезпечує автоматичне виявлення ресурсів і керування ними.

Розроблено програмну модель доступу до апаратного ключа, яка розмежовує функції клієнтів, шлюзу та адаптера до апаратного ключа, а також забезпечує масштабованість, спостережуваність і безпеку.

Визначено метод доступу до ключа в кластерному середовищі, який реалізує серіалізований одночасний доступ, механізми leader election, failover і fencing, що унеможливлюють подвійне використання ліцензії.

Отримані результати демонструють, що запропонований метод дозволяє значно підвищити рівень відмовостійкості, мінімізувати час простою та забезпечити стабільну роботу ліцензійних серверів навіть у випадку часткових

збоїв інфраструктури. Створена модель є універсальною та може бути адаптована для різних типів апаратних ключів і корпоративних середовищ, що робить її практично цінною для впровадження в сучасних системах ліцензування.

3 РЕАЛІЗАЦІЯ ПРОГРАМНОЇ ЧАСТИНИ МЕТОДУ ЗАБЕЗПЕЧЕННЯ ВІДМОВОСТІЙКОСТІ ТА ДОСТУПНОСТІ СЕРВЕРІВ ЛІЦЕНЗУВАННЯ

3.1 Вимоги та засоби реалізації програмного продукту для створення апаратного ключа

Програмне забезпечення апаратного ключа має забезпечувати надійне та захищене виконання криптографічних операцій, ізоляцію секретних даних та стабільну взаємодію із зовнішніми сервісами в контейнеризованому середовищі. Система повинна працювати автономно, гарантувати коректність шифрування й дешифрування, контролювати цілісність апаратного модуля та запобігати несанкціонованому доступу за допомогою механізмів аутентифікації, GPS—верифікації та захисних режимів. Програма має бути здатною до швидкого відновлення після збоїв, підтримувати журналювання подій та внутрішній моніторинг стану пристрою. Основні вимоги діляться на декілька категорій, функціональні, нефункціональні та експлуатаційні вимоги.

Функціональні вимоги:

— виконання операцій шифрування та дешифрування за допомогою AES—GCM або іншого сучасного криптоалгоритму;

— генерація ключів на основі сертифіката та GPS—координат;

— верифікація дозволеної геолокації апаратного ключа перед виконанням криптографічних операцій;

— підтримка захищеного HTTPS—інтерфейсу для взаємодії з зовнішніми сервісами;

— автоматичне переключення у захисний режим у разі некоректного пароля, зміни координат або виявлення втручання;

— генерація правдоподібних фальсифікованих даних у режимі захисту.

Нефункціональні вимоги:

— забезпечення високого рівня безпеки й неможливості вилучення криптографічних ключів із пристрою;

— відмовостійкість та безперервна робота навіть у разі часткових збоїв;

— сумісність із Kubernetes через Device Plugin або мережевий доступ;

— швидкодія, достатня для оброблення багатьох запитів у черзі;

— масштабованість та можливість інтеграції з існуючими сервісами медичної системи.

Експлуатаційні вимоги :

— підтримка журналювання усіх операцій та подій безпеки;

— простота оновлення та обслуговування програмного забезпечення;

— робота в умовах обмежених обчислювальних ресурсів Raspberry Pi 5.

Розроблення програмного забезпечення роботи апаратного ключа є одним із ключових етапів створення методу забезпечення відмовостійкості та доступності серверів ліцензування. Основне завдання програмної частини полягає у забезпеченні безпечного доступу до криптографічних функцій апаратного ключа, а також у забезпеченні коректної роботи механізмів автентифікації, перевірки ліцензій і відновлення доступу у випадку відмови.

Для підвищення стійкості апаратного ключа до несанкціонованого доступу пропонується винесення частини логіки захисного програмного забезпечення безпосередньо на мікрокомп’ютер Raspberry Pi 5. У якості прикладу розглянемо медичний заклад, який зобов’язаний забезпечувати надійне зберігання приватних даних пацієнтів. Оскільки апаратний ключ генерується індивідуально для кожного клієнта з метою максимального захисту його інтелектуальної власності та прав доступу, на нього може бути перенесено частину критично важливих обчислень — зокрема, реалізацію алгоритмів шифрування та дешифрування медичних даних. Структурну схему програмного продукту наведено на рисунку 3.1.

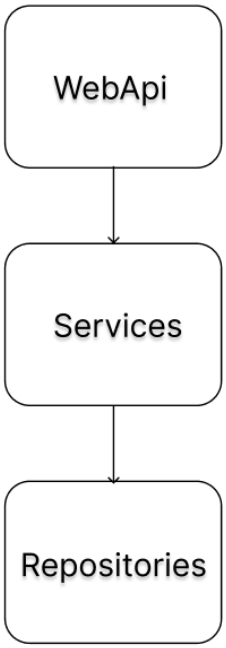


Рисунок 3.1 — Стукрура програмного продукту апаратного ключа

Перший шар — WebApi (або Presentation Layer) — це верхній шар, який приймає HTTP—запити від клієнтів (мобільних застосунків, веб—інтерфейсу, інших сервісів) і повертає їм відповіді. Він відповідає за маршрутизацію, валідацію вхідних даних, авторизацію/аутентифікацію та виклик відповідних сервісів. Його головне завдання — не мати бізнес—логіки, а лише бути «точкою входу» у систему.

Другий шар — Services (або Business Logic Layer) — це серце програми, де зосереджена вся бізнес—логіка. Тут виконується обробка даних, застосування правил, перевірок, алгоритмів, взаємодія між різними частинами домену. Сервісний шар визначає, як саме система повинна поводитися відповідно до вимог бізнесу. Він викликає репозиторії, інші сервіси або інтеграційні клієнти.

Третій шар — Repositories (або Data Access Layer) — це шар доступу до даних. Він відповідає за взаємодію з базою даних або іншими джерелами інформації: читання, запис, оновлення, видалення. Репозиторії приховують деталі реалізації роботи з БД (наприклад, SQL, EF Core, MongoDB), надаючи сервісам чистий інтерфейс під високого рівня.

Разом ці шари створюють структуровану, модульну архітектуру, у якій кожен компонент має чітку зону відповідальності, що спрощує підтримку, тестування та розвиток застосунку.

Оскільки Raspberry Pi 5 — це повноцінний мікрокомп’ютер з процесором ARM і операційною системою Linux Debian, на ньому без проблем можна запускати застосунки, написані на компільованих мовах програмування. На відміну від інтерпретованих мов, компільовані дають кращу продуктивність, стабільність і контроль над апаратними ресурсами, що особливо важливо для пристроїв, які виконують криптографічні операції, працюють із сенсорами або забезпечують високу надійність роботи. Завдяки цьому Raspberry Pi ідеально підходить для запуску застосунків, зібраних на .NET, оскільки .NET повністю підтримується в Linux—середовищі та оптимізований для ARM—архітектури [26].

C# у поєднанні з ASP.NET Core є одним із найкращих виборів для такого пристрою. C# забезпечує високу швидкість роботи, сильну типізацію, потужні можливості роботи з пам’яттю та вбудовані засоби для криптографії й безпеки. ASP.NET Core, у свою чергу, дозволяє створити легкий, продуктивний та кросплатформений веб—сервіс, який може працювати прямо на Raspberry Pi, обробляючи запити, керуючи апаратними ключами або взаємодіючи з зовнішніми системами. Разом вони дають змогу побудувати на Raspberry Pi 5 надійну серверну платформу, яка поєднує в собі простоту розробки та промислову якість виконання.

Середовище розробки Visual Studio було обрано як основна IDE для створення програмного забезпечення, що взаємодіє з апаратним ключем, завдяки своїй широкій функціональності, інтеграції з екосистемою .NET та високому рівню підтримки інструментів для побудови, тестування й налагодження застосунків. Visual Studio забезпечує повний цикл розробки — від написання й структурування коду до автоматизованого деплою у контейнеризовані середовища, що є критично важливим у межах реалізації програмно—апаратного комплексу (рис 3.2). Вбудовані інструменти профілювання,

інтеграція з Git, підтримка контейнерів Docker, можливість роботи з NuGet—пакетами та розширені можливості відлагодження значно спрощують процес розробки і дозволяють оперативно виявляти помилки на ранніх етапах. Крім того, Visual Studio забезпечує зручне керування залежностями, автоматичну генерацію сертифікатів для HTTPS, а також підтримує модульне тестування, що робить її оптимальним вибором для побудови надійного та безпечного серверного програмного забезпечення, яке взаємодіє з апаратними засобами [27].

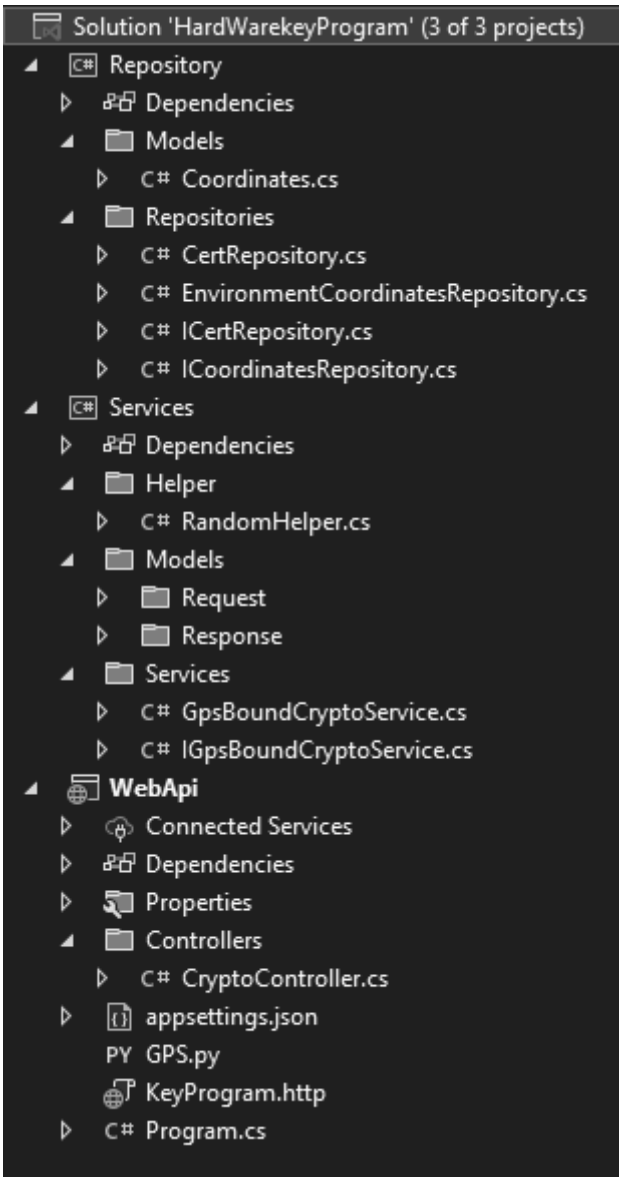


Рисунок 3.2 — Програмний комплекс для апартного ключа

Додатковий рівень захисту забезпечує інтегрований GPS—модуль (лістинг 3.1), який отримує географічні координати поточного розташування пристрою. На основі координат та введеного адміністратором пароля апаратний

ключ виконує процеси шифрування та дешифрування даних. У разі зміщення пристрою та зміни координат він продовжує роботу, однак результати шифрування будуть відрізнятися від початкових, що створює додатковий захисний бар’єр проти атак, пов'язаних з фізичним копіюванням чи підміною ключа.

Лістинг 3.1 — програма отримання координат від супутника через GPS—модуль

```
import serial
import time
import string
import pynmea2

while True:
  port="/dev/ttyS2"
  ser=serial.Serial(port, baudrate=9600, timeout=0.5)
  dataout = pynmea2.NMEAStreamReader()
  newdata=ser.readline().decode('unicode_escape')
  if newdata[0:6] == "$GPRMC":
    newmsg=pynmea2.parse(newdata)
    lat=newmsg.latitude
    lng=newmsg.longitude
    gps = "(широта, довгота): (" + str(lat) + "," + str(lng) + ")"
    print(gps)
```

Якщо ж адміністратор вводить некоректний пароль, пристрій автоматично переходить у режим захисту: замість реальних даних він генерує правдоподібні, але фальсифіковані результати. Це суттєво ускладнює спроби злому, оскільки система не лише блокує доступ, а й створює додаткові інформаційні пастки для потенційного зловмисника.

У випадку, коли отримати коректні GPS—координати (встановимо погрішність до 50м) зі супутника неможливо (наприклад, через екранування сигналу або недостатню кількість видимих супутників), пристрій активує резервний режим роботи. У цьому режимі програмна логіка генерує правдоподібні, але фальсифіковані координати та пов’язані з ними результати обчислень, що унеможливлює отримання реальних даних сторонньою особою.

Після підключення апаратного ключа до живлення доступ до функціоналу здійснюється лише після введення адміністративного пароля. У разі введення некоректного пароля пристрій навмисно повертає неправдиві, але зовні достовірні результати. Такий підхід підсилює захист від несанкціонованого доступу, оскільки зловмисник не може визначити, чи є отримані дані коректними, а сама система унеможливлює зчитування або відтворення справжніх алгоритмів і вихідних значень.

3.2 Побудова архітектури програмного продукту для захисту апаратним ключем

Проєктування програмного забезпечення для медичного закладу потребує підвищеного рівня информаційної безпеки, оскільки система працює з чутливою інформацією, персональними та медичними даними пацієнтів. У межах розробленого рішення запропоновано архітектуру, у якій усі операції шифрування та дешифрування виконуються за допомогою апаратного ліцензійного ключа, що фізично розміщується у захищеному сегменті інфраструктури лікарні. Це дозволяє повністю виключити можливість читання чи модифікації даних поза межами довіреного апаратного середовища.

Користувацький інтерфейс програмного комплексу (рис 3.1) побудований на основі REST—архітектури та документований відповідно до стандарту OpenAPI 3.0, що забезпечує прозору взаємодію між клієнтськими застосунками медичного закладу та серверною частиною системи.

Додатково в архітектурі реалізовано механізми автентифікації та контролю доступу до криптографічних сервісів, що базуються на захищеному протоколі HTTPS із використанням сертифікатів та ролей доступу. Усі запити до апаратного ключа проходять через сервер ліцензування, який виконує функції централізованої маршрутизації, журналювання та серіалізації операцій, що унеможливлює одночасний неконтрольований доступ до апаратного ресурсу. Такий підхід забезпечує відповідність вимогам до захисту медичних даних, підвищує відмовостійкість системи та дозволяє зберігати цілісність і

конфіденційність інформації навіть у разі відмови окремих компонентів або підвищеного навантаження.

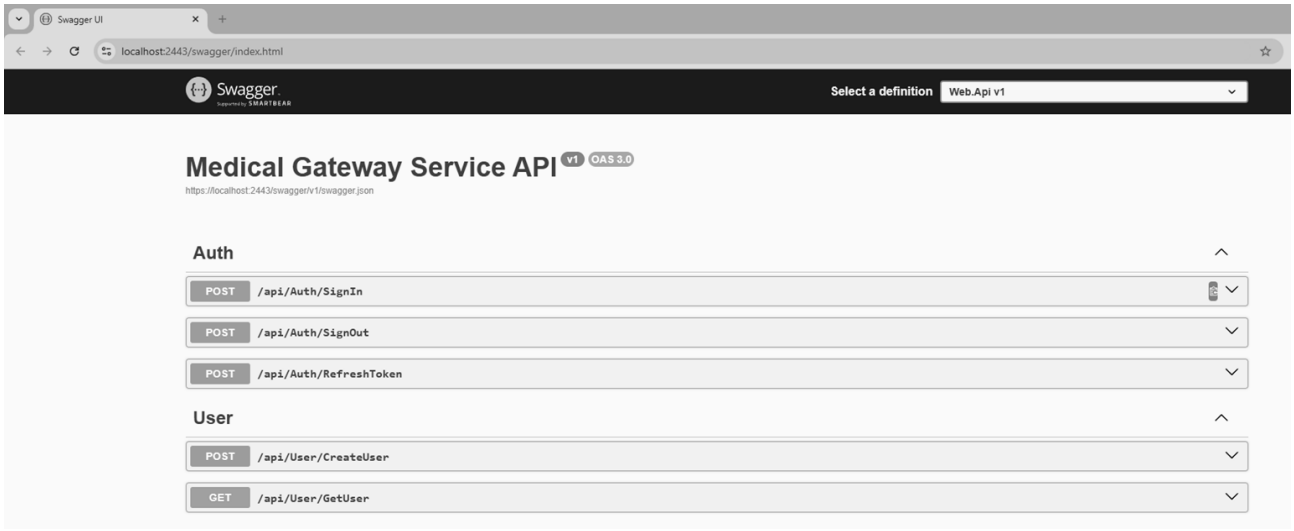


Рисунок 3.3 — інтерфейс програмного засобу

Перелік ендпоінтів Medical Gateway Service API та їх функціональне призначення:

— POST /api/Auth/SignIn, виконує аутентифікацію користувача за логіном і паролем, генерує та повертає JWT—токен доступу та refresh—токен, використовується клієнтськими системами медзакладу для входу в систему;

— POST /api/Auth/SignOut, завершує активну сесію користувача, анулює refresh—токен та блокує подальше використання старих токенів, забезпечує коректне завершення роботи в межах медичної інформаційної системи;

— POST /api/Auth/RefreshToken, оновлює пару токенів access + refresh для продовження сесії без повторного входу, використовується при тривалих робочих сесіях медичного персоналу, підвищує зручність та безпеку доступу до даних пацієнтів;

— POST /api/User/CreateUser, створює нового користувача у системі (лікаря, медсестру, адміністратора тощо), виконує валідацію введених даних та призначення ролей доступу, застосовується адміністраторами медичного закладу для управління персоналом;

— GET /api/User/GetUser, повертає інформацію про користувача за його унікальним ідентифікатором, реалізує захищений доступ до профілів співробітників та їхніх привілеїв, використовується для відображення даних персоналу та перевірки їх прав.

Переваги інтерфейсу API:

— чітка модульність окрема група ендпоінтів для аутентифікації та керування користувачами;

— безпечна взаємодія усі виклики виконуються через HTTPS і захищені токенами;

— інтеграція з апаратним ключем під час операцій створення або отримання даних пацієнтів система викликає апаратний криптомодуль для шифрування/дешифрування;

— стандартизований формат OpenAPI 3.0 забезпечує автоматичну генерацію клієнтів і документації.

Основною ідеєю є винесення частини критично важливих обчислень на апаратний модуль Raspberry Pi 5, який виконує криптографічні операції локально, забезпечуючи їхню апаратну захищеність. Апаратний ключ працює як мережевий криптопроцесор і надає сервіс шифрування через захищене HTTPS—з'єднання на адресі https://localhost:5443 у межах хост—системи. Програмний продукт лікарні взаємодіє з апаратним ключем через окремий мікросервіс — CryptoGatewayService, який виступає проксі між бізнес—логікою застосунку та функціоналом апаратного ключа.

Під час збереження даних пацієнта система надсилає HTTP—запит до апаратного ключа, передаючи дані у незашифрованому вигляді. Апаратний модуль виконує операцію шифрування, формує криптограму та повертає її у вигляді структури JSON. Зворотна операція дешифрування, виконується за тим самим принципом: застосунок передає зашифрований блок на ключ, який відновлює оригінальний вміст. Такий підхід гарантує, що відкриті дані ніколи не покидають захищеного середовища апаратного модуля [28].

Лістинг 3.2 — Приклад сервісу який шифрує та дешифрує дані, звертаючись до апаратного ключа.

```
using System.Net.Http;
using System.Net.Http.Json;
using System.Security.Authentication;
using System.Text.Json;
public class CryptoGatewayService
{
   private readonly HttpClient _httpClient;
   public CryptoGatewayService()
   {
      var handler = new HttpClientHandler
      {
         SslProtocols = SslProtocols.Tls12,
         ClientCertificateOptions = ClientCertificateOption.Manual,
         ServerCertificateCustomValidationCallback =
            (sender, cert, chain, errors) => true // довіряємо локальному ключу
      };

      _httpClient = new HttpClient(handler)
      {
         BaseAddress = new Uri("https://localhost:5443/")
      };
   }
   public async Task<string> EncryptAsync(object data)
   {
      var response = await _httpClient.PostAsJsonAsync("encrypt", data);
      response.EnsureSuccessStatusCode();

      var json = await response.Content.ReadAsStringAsync();
      return json;
   }
   public async Task<string> DecryptAsync(string encryptedPayload)
   {
      var content = JsonContent.Create(new { payload = encryptedPayload });
      var response = await _httpClient.PostAsync("decrypt", content);
      response.EnsureSuccessStatusCode();
      var json = await response.Content.ReadAsStringAsync();
      return json;
   }
}
```

Роль сервісу в архітектурі медичного застосунку полягає в такому:

— шифрування медичних карт пацієнтів перед збереженням у базу даних;

— дешифрування даних при запиті від уповноважених лікарів;

— неможливість доступу до відкритих даних без фізичного доступу до апаратного ключа;

— верифікація операцій криптографії через TLS і локальні сертифікати;

— ізоляція бізнес—логіки від криптографічних реалізацій.

Таким чином, навіть у разі компрометації бази даних або одного з мікросервісів, зловмисник не зможе відновити пацієнтські дані без апаратного ключа, GPS—прив’язки та правильної аутентифікації.

3.3 Розроблення алгоритму доступу до апаратного ключа в контейнеризованому середовищі

Забезпечення коректного, безпечного та відмовостійкого доступу до апаратного ключа в контейнеризованому середовищі є критично важливим аспектом функціонування розроблюваної системи. На відміну від програмних сервісів, які можуть горизонтально масштабуватися у межах Kubernetes—кластера, апаратний ключ є фізично унікальним ресурсом, здатним виконувати лише одну криптографічну операцію за одиницю часу. Це створює асиметрію між обчислювальними можливостями кластера та обмеженістю апаратного пристрою, що потребує спеціального алгоритму доступу та механізму серіалізації запитів. [29]

На рисунку 3.4 структуру представлено доступу до апаратного криптографічного ключа в контейнеризованому середовищі, яка складається з трьох основних компонентів: апаратного ключа, Kubernetes—кластера та сервісу License Gateway. Апаратний ключ реалізований у вигляді окремого фізичного пристрою, оснащеного мережевим інтерфейсом Ethernet та GPS—модулем, який підключено до інфраструктури лікарні. Через захищений канал Crypto HTTPS—пристрій приймає запити на шифрування та дешифрування.

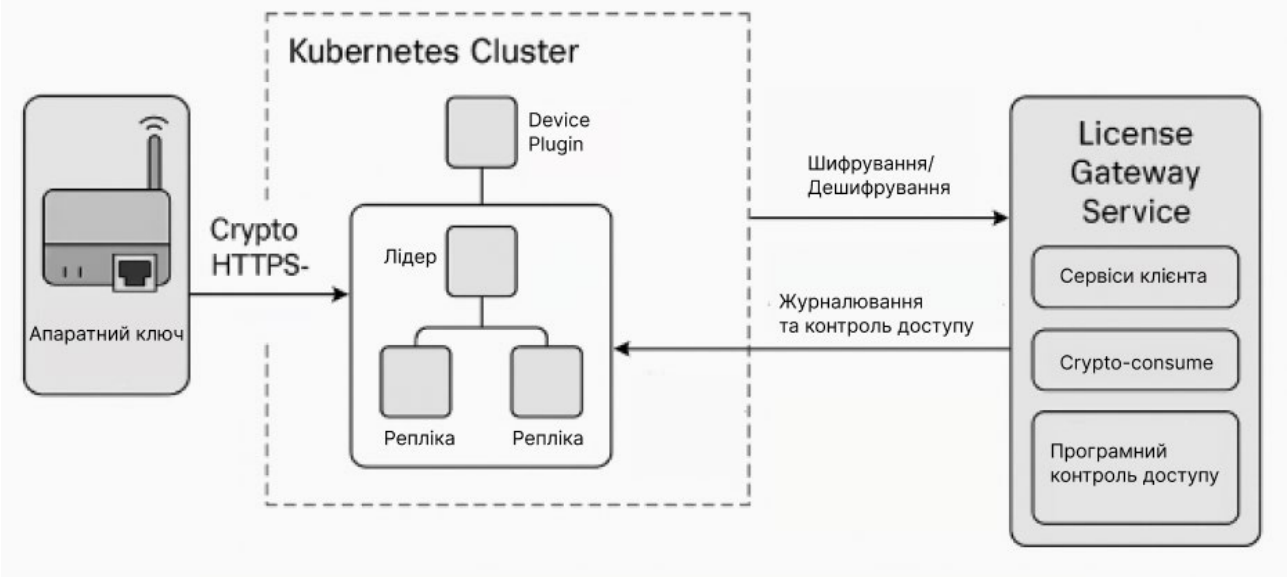


Рисунок 3.4 — Схема доступу до апаратного ключа в контейнеризованому середовищі Kubernetes

У межах Kubernetes—кластеру працює Device Plugin, який позначає вузли, що мають доступ до апаратного ключа. У кластері застосовується механізм вибору лідера (leader election): один вузол виконує роль «лідера» та має ексклюзивний доступ до криптографічного модуля, тоді як інші вузли працюють як «репліки» та можуть автоматично перебрати роль лідера у разі відмови активного вузла. License Gateway Service розміщений праворуч і виконує функції маршрутизації, шифрування/дешифрування, журналювання та перевірки доступу. Він отримує запити від сервісів клієнтів та передає їх до лідера кластеру, який взаємодіє з апаратним ключем. [30]

На рисунку також показано логічні компоненти License Gateway: модуль клієнтських сервісів, компонент Crypto—consume, що відповідає за серіалізацію криптографічних операцій, та модуль програмного контролю доступу (RBAC + журналювання). Завдяки такій архітектурі всі криптографічні операції залишаються всередині контрольованого середовища, а відкриті дані ніколи не покидають меж довіреної інфраструктури.

Алгоритм взаємодії між компонентами можна подати через призму моделі OSI:

— physical Layer (1—й рівень) апаратний ключ фізично під’єднано до мережевої інфраструктури датацентру через Ethernet;

— data Link Layer (2—й рівень) комунікація виконується через стандартні Ethernet—фрейми; пристрій отримує MAC—адресу та бере участь у локальній мережі;

— network Layer (3—й рівень) ключ отримує IP—адресу (DHCP або статичну), після чого стає доступним для внутрішніх сервісів;

— transport Layer (4—й рівень) взаємодія здійснюється через TCP, що гарантує надійність доставки криптографічних запитів;

— session Layer (5—й рівень) створюється стійкий HTTPS—сеанс між License Gateway та апаратним ключем. TLS забезпечує узгодження параметрів безпеки;

— presentation Layer (6—й рівень) дані серіалізуються у формат JSON, а шифрування здійснюється на апаратному ключі з використанням AES—GCM та AAD;

— application Layer (7—й рівень) запити приймає License Gateway, який визначає тип операції (encrypt/decrypt), перевіряє доступ, журналює події та надсилає дані до ключа.

Спрощена реалізація: License Gateway на одному сервері

Для зменшення кількості інфраструктурних компонентів та спрощення розробки систему можна розгорнути таким чином, що License Gateway працюватиме, на тій самій машині, де знаходиться клієнтський сервіс, або на тому ж вузлі, що й лідер Kubernetes.

У такому випадку мережеві затримки мінімальні, а архітектура стає простішою для налагодження. Це підходить для тестових середовищ або для невеликих медичних закладів без складної кластерної інфраструктури.

У попередньому розділі реалізовано Medical License Gateway — серверний модуль лікарської інформаційної системи, який запускається в

контейнеризованому середовищі й маршрутизує всі операції шифрування через апаратний ключ. Gateway виконує:

— шифрування медичних карт пацієнтів перед збереженням у базу даних;

— дешифрування даних перед передачею лікарю;

— автентифікацію та авторизацію кожного запиту;

— журналювання дій персоналу;

— перевірку сертифікатів та геолокації апаратного ключа.

License Gateway у рамках цього кейсу забезпечує безпечну взаємодію між медичною інформаційною системою та апаратним криптомодулем, гарантуючи, що відкриті дані ніколи не залишають довіреного середовища. Такий підхід дозволяє серйозно підвищити рівень безпеки медичних систем, унеможливлюючи витік або компрометацію чутливої інформації.

3.4 Забезпечення захисту програмного засобу за допомогою апаратного ключа

Захист програмного забезпечення медичного закладу базується на поєднанні апаратних, криптографічних та програмних механізмів, що утворюють багаторівневу модель безпеки. Центральним компонентом системи є апаратний ключ, який виконує роль захищеного криптографічного модуля та надає функції шифрування, дешифрування й перевірки автентичності даних. Винесення критично важливих обчислень на апаратний пристрій забезпечує апаратну ізоляцію секретних ключів, мінімізуючи ризики витоку інформації у разі компрометації програмного середовища або бази даних.

Одним із ключових аспектів забезпечення безпеки є виконання всіх криптографічних операцій виключно в апаратному модулі Raspberry Pi 5. Програмні сервіси передають дані на шифрування через захищений HTTPS—інтерфейс (`https://<ip—ключа>:5443`), тоді як сам модуль використовує локальні сертифікати для підтвердження легітимності запиту. Таким чином,

відкриті медичні дані ніколи не покидають меж довіреного апаратного середовища. Під час збереження даних пацієнта система викликає апаратний модуль, який формує криптограму та повертає її у стандартизованому форматі. Аналогічний механізм використовується для дешифрування — бізнес—логіка працює лише з даними, які надходять від апаратного ключа.

Важливим елементом захисту є інтегрована GPS—верифікація, яка забезпечує географічну прив'язку криптографічних операцій. Апаратний ключ активує GPS—модуль NEO—8M та перевіряє, що координати пристрою відповідають дозволеній геозоні медичного закладу. Наприклад, місцем розташування центру безпеки може бути координата (49.233987, 28.468133). Якщо пристрій знаходиться поза межами дозволеного радіусу або не може отримати сигнал супутника — він автоматично переходить у захисний режим. У захисному режимі криптографічні операції не виконуються, а замість реальних результатів генеруються правдоподібні, але повністю фальсифіковані дані, що істотно ускладнює можливі спроби реверс—інжинірингу [31].

Додатковим механізмом захисту є перевірка автентичності адміністратора. Під час підключення до апаратного ключа адміністратор зобов'язаний ввести локальний пароль, який є частиною внутрішнього механізму доступу. Якщо пароль введено некоректно, ключ переходить у режим активної протидії: у системний лог надсилається сповіщення про спробу несанкціонованого доступу, структура операцій криптографії спотворюється, а весь подальший функціонал повертає фальсифіковані дані (лістинг 3.3). Така поведінка створює додатковий рівень безпеки, перетворюючи пристрій на «інтелектуальну пастку» для зловмисників.

Лістинг 3.3 — Приклад переходу програми у захисний режим

```
private static volatile bool _isInSecureState = false;
private void EnsureWithinAllowedRadius(Coordinates currentLocation){
    var referenceLocation = _coordinatesRepository.GetDefaultCoordinates();
    // Update the secure state based on current location; false when inside allowed radius.
    _isInSecureState = !IsWithinRadius(currentLocation, referenceLocation, EnvironmentCoordinatesRepository.AllowedRadiusMeters);
```

```
    }

        private static bool IsWithinRadius(Coordinates current, Coordinates
reference, double radiusMeters)
    {
        double distance = HaversineDistanceMeters(current.Lat, current.Lon,
reference.Lat, reference.Lon);
        return distance <= radiusMeters;
    }
    if (_isInSecureState)
    {
        // Return a fake payload when device is in insecure state
        return Task.Run(RandomHelper.GenerateFakeEncryptedPayload);
    }
    if (_isInSecureState)
    {
        return                                Task.Run(()                                =>
RandomHelper.GenerateFakeDecryptedData(payload));
    }
```

Не менш важливими є програмні механізми контролю доступу всередині кластерного середовища. Взаємодія між мікросервісами та апаратним ключем відбувається через Medical Gateway Service API, який виконує аутентифікацію, авторизацію, ведення журналу доступу та серіалізацію криптографічних операцій. Завдяки цьому навіть у середовищі з кількома вузлами та десятками сервісів забезпечується контрольований, впорядкований та безпечний доступ до апаратного ключа. Система гарантує, що лише дозволені запити з валідними токенами можуть взаємодіяти з апаратним криптомодулем.

Таким чином, інтеграція апаратного ключа у програмний комплекс медичного закладу забезпечує багаторівневий захист, який охоплює апаратну ізоляцію секретів, географічну прив’язку, сертифікатну автентифікацію, контроль доступу на рівні кластерної інфраструктури та механізми протидії нелегітимному втручанню. Комбінація цих методів дозволяє створити систему, стійку до фізичних, мережевих та програмних атак, що є критично важливим для роботи з медичною інформацією та персональними даними.

Отже, у процесі дослідження було розроблено комплексний підхід до створення та інтеграції апаратного ключа у сучасне контейнеризоване середовище. Запропоноване рішення поєднує апаратні та програмні механізми захисту, що дозволяє забезпечити високий рівень безпеки, відмовостійкості та ізоляції критично важливих обчислень. Використання апаратного модуля на базі Raspberry Pi 5 дозволило винести криптографічні операції за межі прикладного програмного забезпечення, унеможливлюючи доступ до секретних ключів та відкритих даних сторонніми службами.

Розроблена архітектура ґрунтується на застосуванні мережевої взаємодії через захищений HTTPS—протокол, механізмів сертифікатної автентифікації, географічної прив’язки на основі GPS—координат, а також контейнеризованих рішень Kubernetes. Це забезпечує як масштабованість програмних сервісів, так і контрольований доступ до унікального апаратного ресурсу. Важливою складовою моделі є механізми серіалізації запитів, вибору лідера у кластері та автоматичного відновлення у випадку відмов, що гарантує стабільність та безперервність роботи системи [32].

Запропонована система також передбачає функції активної протидії зловмисним діям: у разі спроби компрометації пристрій переходить у захисний режим, спотворює результати криптографічних операцій і ініціює повідомлення адміністраторам. Такий підхід істотно підвищує стійкість до атак, реверс—інжинірингу та фізичного втручання. У результаті сформовано цілісну, надійну та технологічно сучасну модель захисту даних, що може використовуватися в медичних та інших критично важливих інформаційних системах, де конфіденційність та безпека мають визначальне значення.

4. ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНА ПЕРЕВІРКА

4.1 Методика тестування та опис експериментального середовища

Методика тестування розробленого методу забезпечення відмовостійкості та доступності серверів ліцензування визначає порядок, умови, програмно—апаратну конфігурацію та критерії оцінювання роботи системи. Вона дає змогу на практиці підтвердити працездатність алгоритмів доступу до апаратного ключа у контейнеризованому середовищі та оцінити стабільність функціонування всіх компонентів при різних сценаріях навантаження та відмов [33].

Для проведення експериментальних досліджень було підготовлено окремий тестовий стенд, що за структурою та характеристиками максимально наближається до умов промислової експлуатації. Основна ідея експериментального середовища полягала у повторенні типової корпоративної архітектури, де сервери ліцензування працюють у Kubernetes—кластері, а доступ до апаратних ключів організований через контейнеризований сервіс—шлюз.

Тестування розробленої системи здійснювалося в умовах локального експериментального середовища, яке моделює роботу апаратного криптографічного модуля та Medical Gateway у наближених до реальних умовах. Основною платформою для проведення експериментів виступав локальний персональний комп’ютер, на якому було розгорнуто як програмну частину апаратного ключа, так і серверний модуль Medical Gateway. Для забезпечення відповідності апаратним обмеженням реального пристрою виконання програми криптомодуля було штучно обмежено до 4 GB оперативної пам’яті — аналогічного обсягу, який використовується на Raspberry Pi 5. Це дозволило достовірно відтворити поведінку системи в умовах обмежених ресурсів і оцінити стабільність роботи під час навантажень.

Medical Gateway розгортався у вигляді окремого локального сервісу, який взаємодіє з криптомодулем через захищений HTTPS—інтерфейс. Для моделювання реальних сценаріїв обміну даними було застосовано локальне мережеве середовище, що дозволило протестувати затримки, пропускну

здатність та відмовостійкість. Окрему увагу приділено тестуванню поведінки системи у випадку помилкової автентифікації, некоректних сертифікатів, втрати мережевого підключення та імітації зміни GPS—координат.

Для кількісної оцінки продуктивності використовувалися стандартні інструменти навантажувального тестування та benchmark—тести, що включали вимірювання часу виконання операцій шифрування та дешифрування, максимальну кількість оброблених запитів за секунду, затримки при серіалізації запитів та стабільність роботи системи під довготривалим навантаженням. Також проводилися стрес—тести, у рамках яких система отримувала значно вищу інтенсивність запитів, ніж у реальних умовах, з метою визначення меж її продуктивності та стабільності [34].

Експериментальне середовище дозволило комплексно оцінити роботу розробленого рішення, визначити сильні сторони, можливі вузькі місця, а також підтвердити коректність реалізації механізмів автентифікації, GPS—верифікації, журналювання та оброблення помилок. Отримані результати стали основою для формування висновків та рекомендацій щодо подальшої оптимізації системи [35].

4.2 Перевірка працездатності та відмовостійкості розробленого методу

Перевірка працездатності розробленого методу здійснювалася як на рівні інтеграційного тестування (рис 4.1), так і шляхом модульної перевірки основних компонентів системи. Зокрема, особлива увага приділялася функціонуванню механізму географічної верифікації, генерації захисних фальсифікованих даних, логіці шифрування та дешифрування, а також відповідності поведінки системи умовам відмовостійкості. Для цього було розроблено низку юніт—тестів, які моделюють різні реальні та аварійні сценарії. Тестове середовище базувалося на локальному комп’ютері, де було обмежено використання оперативної пам’яті до 4 GB — аналогічно умовам реального пристрою. Також застосовувалися інструменти навантажувального тестування (benchmark), що дозволили оцінити

часові характеристики криптографічних операцій та поведінку системи під дією пікових навантажень.

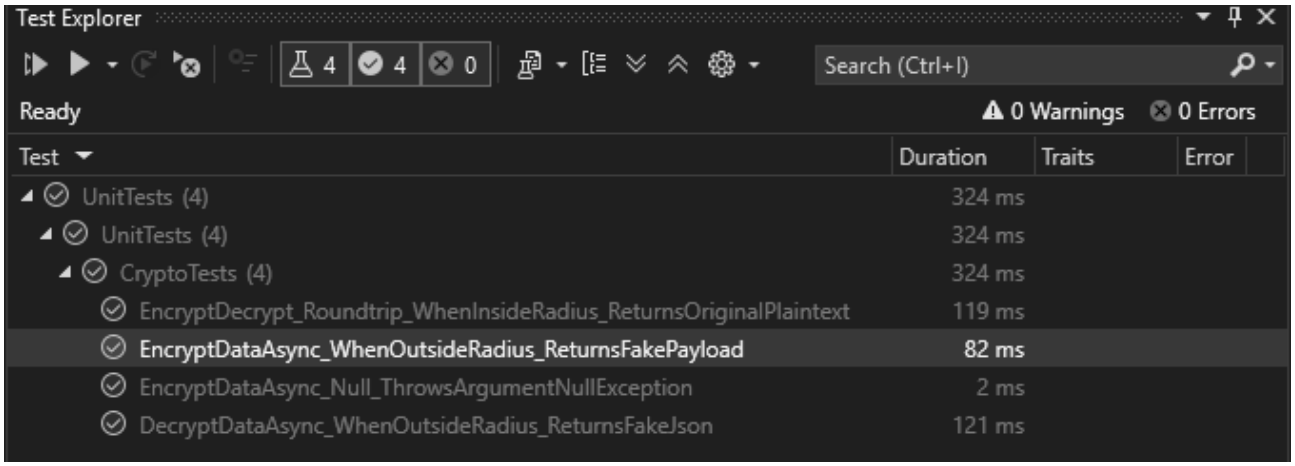


Рисунок 4.1 — інтеграційні тести

У першому тесті EncryptDecrypt_Roundtrip_WhenInsideRadius_ReturnsOriginalPlaintext перевіряє базову коректність роботи системи у штатному режимі. Якщо поточні GPS—координати збігаються з дозволеними, сервіс повинен виконати повноцінне шифрування та дешифрування даних, після чого результат дешифрування має повністю збігатися з оригінальним JSON. Цей тест підтверджує правильність реалізації AES—GCM, формування AAD, генерації ключів, а також відповідність логіки перетворень криптографічним стандартам.

Другий тест EncryptDataAsync_WhenOutsideRadius_ReturnsFakePayload перевіряє механізм переходу системи в захисний режим у разі, коли GPS—координати вказують на те, що пристрій знаходиться поза дозволеною геозоною. У цьому випадку сервіс шифрування не виконує реальну криптографічну операцію, а повертає фальсифікований результат — випадкові за структурою, але коректні за формою дані (ciphertext довжиною 32 байти, nonce — 12 байт, tag — 16 байт). Така поведінка унеможливлює реверс—інжиніринг алгоритму з боку зловмисника та підтверджує стійкість алгоритму до фізичного переміщення пристрою або спроб маніпуляції координатами.

У третьому тесті EncryptDataAsync_Null_ThrowsArgumentNullException перевіряється коректність оброблення некоректного вхідного параметра під час шифрування. Передавання `null` повинно призводити до генерації винятку типу `ArgumentNullException`, що гарантує коректність валідації даних на ранньому етапі та запобігає неконтрольованому виконанню криптографічної операції. Це моделює ситуацію, за якої сервіс може отримати порожній запит від некоректного клієнта або внаслідок помилки у бізнес—логіці.

У четвертому тесті DecryptDataAsync_WhenOutsideRadius_ReturnsFakeJson перевіряється аналогічний механізм, але вже для операції дешифрування. У випадку виходу за межі геозони сервіс повинен повернути синтетичний JSON—документ, який містить структурно коректні дані (`patientId`, `name` тощо), але не має відношення до реального вмісту. Це підтверджує працездатність алгоритму активної протидії, який не лише блокує операції, але й надає фальсифіковані результати, створюючи додатковий рівень криптографічної непередбачуваності.

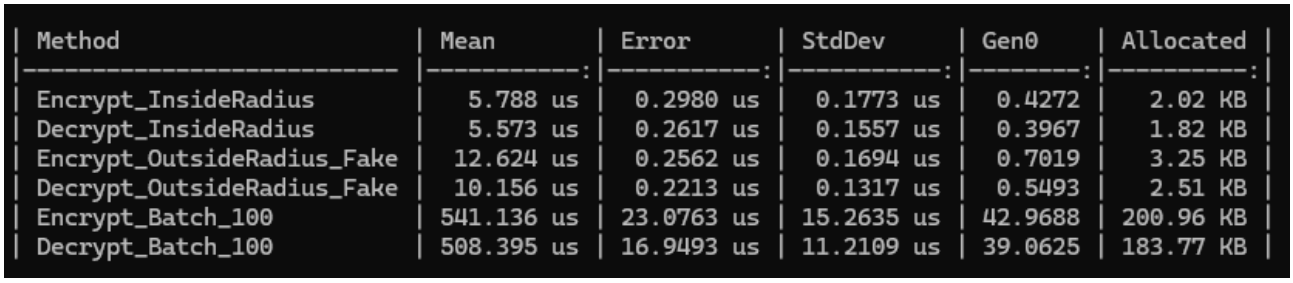
Завдяки проведеним тестам підтверджено, що система працює стабільно в нормальних умовах, адекватно реагує на відмови та переходить у захисний режим у випадку порушення політик безпеки. Набір тестів демонструє здатність сервісу коректно обробляти критичні ситуації, забезпечувати ізоляцію відкритих даних та унеможливлювати витік ключів. Таким чином, розроблений метод володіє високою відмовостійкістю та забезпечує захисні властивості, необхідні для використання у медичних інформаційних системах.

4.3 Навантажувальне тестування та оцінка продуктивності

Для оцінки продуктивності розробленого сервісу `GpsBoundCryptoService` було проведено серію навантажувальних тестів із використанням бібліотеки BenchmarkDotNet. Метою експерименту було визначення середнього часу виконання операцій шифрування та дешифрування в штатному режимі (всередині дозволеного GPS—радіуса), а також у захисному режимі (поза радіусом), де сервіс повертає фальсифіковані результати. Окремо оцінювалася

продуктивність при пакетній обробці 100 послідовних запитів та обсяги виділення пам’яті під час виконання операцій [36].

Результати тестування наведено на рисунку 4.2



Method	Mean	Error	StdDev	Gen0	Allocated
Encrypt_InsideRadius	5.788 us	0.2980 us	0.1773 us	0.4272	2.02 KB
Decrypt_InsideRadius	5.573 us	0.2617 us	0.1557 us	0.3967	1.82 KB
Encrypt_OutsideRadius_Fake	12.624 us	0.2562 us	0.1694 us	0.7019	3.25 KB
Decrypt_OutsideRadius_Fake	10.156 us	0.2213 us	0.1317 us	0.5493	2.51 KB
Encrypt_Batch_100	541.136 us	23.0763 us	15.2635 us	42.9688	200.96 KB
Decrypt_Batch_100	508.395 us	16.9493 us	11.2109 us	39.0625	183.77 KB

Рисунок 4.2 — Результати лоад тестування

У штатному режимі (коли поточні координати знаходяться всередині дозволеного радіуса) середній час виконання однієї операції шифрування становить близько менше ніж 6 мікросекунд, а дешифрування — близько менше ніж 6 мікросекунд, що є достатньо низькими значеннями для використання в медичних інформаційних системах із типовими навантаженнями.

Обсяг динамічно виділеної пам’яті на одну операцію знаходиться на рівні приблизно 2 кілобайти, що свідчить про помірне використання ресурсів і відсутність надмірних алокацій, які могли б призводити до частих спрацювань сміттєзбирача.

Режим роботи поза дозволеним GPS—радіусом, у якому сервіс повертає фальсифіковані дані, очікувано є дещо дорожчим за часом та пам’яттю: шифрування виконується в середньому за близько 13 мікросекунд, а дешифрування — за близько 10 мікросекунд при збільшенні обсягів виділеної пам’яті до приблизно 3,25 KB і приблизно 2,5 KB відповідно. Це зумовлено необхідністю генерації синтетичних даних, формування фейкових структур та додаткових обчислень, пов’язаних із режимом захисту. Водночас навіть у захисному режимі час відповіді залишається в межах десятків мікросекунд, що не створює критичних затримок для прикладних сервісів.

Пакетні тести на 100 операцій продемонстрували квазі—лінійне масштабування часу виконання: обробка 100 послідовних операцій шифрування

займає в середньому трохи більше ніж пів мілісекунди, а 100 операцій дешифрування — трохи менше ніж пів мілісекунди. Це узгоджується з індивідуальними результатами (приблизно від 5 до 6 мкс на одну операцію) та підтверджує відсутність прихованих «вузьких місць» чи блокувань, які могли б призводити до деградації продуктивності при серійному виконанні запитів. Обсяг виділеної пам’яті для пакетної обробки складає близько 200 KB на 100 операцій, що також відповідає масштабуванню на рівні 2 KB на одну операцію.

Отримані результати свідчать про те, що розроблений сервіс GpsBoundCryptoService забезпечує низькі затримки та передбачувану продуктивність як у нормальному режимі, так і в режимі захисту. Час відповіді в межах десятків мікросекунд дозволяє інтегрувати сервіс у мікросервісну архітектуру медичного закладу без суттєвого впливу на загальну затримку оброблення запитів. Лінійне масштабування при пакетних тестах і помірний рівень виділення пам’яті підтверджують можливість використання методу в умовах реальних навантажень, а також його придатність для подальшого розширення і розгортання в кластерному середовищі Kubernetes.

4.4 Аналіз результатів експериментів та висновки

У ході експериментальних досліджень було проведено комплексне тестування працездатності, продуктивності та відмовостійкості розробленої системи, що поєднує програмний компонент Medical Gateway та апаратний криптографічний модуль. Основною метою експериментів було підтвердити коректність реалізації криптографічних операцій, здатність системи до роботи в умовах обмежених ресурсів, а також перевірити стійкість до відмов і некоректних умов експлуатації.

Результати навантажувального тестування продемонстрували, що операції шифрування та дешифрування у дозволеній геозоні виконуються в середньому від 5,7 мкс до 5,8 мкс, що забезпечує мінімальні затримки та дозволяє інтегрувати систему у реальні медичні сервіси без відчутного впливу на швидкодію. Обсяг пам’яті, що виділяється під одну операцію (близько 2 KB), є

стабільним та не призводить до надмірних навантажень на систему збору сміття. Це свідчить про оптимальність реалізації криптографічних алгоритмів та відсутність невиправданих алокацій.

У режимі захисного виконання—коли пристрій знаходиться поза межами дозволеного радіусу—витрати часу очікувано зростають: Encrypt_OutsideRadius_Fake у середньому виконується приблизно за 12,6 мкс, а Decrypt_OutsideRadius_Fake — за 10,1 мкс. Це пов’язано з додатковими обчисленнями, необхідними для генерації фальсифікованих результатів. Водночас такі затримки залишаються незначними та не впливають критично на загальну роботу системи. Важливо підкреслити, що генерація фейкових даних відбувається структуровано, з формуванням правдоподібних JSON—об’єктів, що унеможливлює розкриття інформації про внутрішні алгоритми за межами дозволеної геолокації.

Пакетні тести підтвердили лінійне масштабування: виконання 100 операцій шифрування та дешифрування займає відповідно за 541 мкс та 508 мкс, що корелює зі значеннями одиничних викликів. Це свідчить про відсутність прихованих блокувань, конфліктів або деградації продуктивності при збільшенні навантаження. Система поводиться прогнозовано, а час виконання прямо пропорційний кількості операцій, що є важливим для побудови сервісів з високою частотою транзакцій.

Окремо проведено функціональні тести, що підтвердили правильність переходу в захисний режим при порушенні політик безпеки: зміна координат, некоректний пароль адміністратора або некоректний формат даних викликають генерацію фейкових криптограм та попереджень, не допускаючи розкриття ключового матеріалу або реальних даних. Такі механізми відповідають вимогам до апаратної криптографії та підвищують загальну стійкість системи до фізичних і програмних атак [37].

Таким чином, результати експериментального дослідження підтверджують, що розроблений метод забезпечує високу продуктивність, низькі затримки, передбачувану поведінку під навантаженням та ефективні механізми

захисту. Система є відмовостійкою, коректно обробляє критичні ситуації, не допускає витоку відкритих даних та демонструє стабільність навіть у випадку некоректних умов експлуатації. Отримані результати свідчать про те, що запропонований підхід може бути надійно інтегрований у медичні інформаційні системи та інші критично важливі інфраструктури, де безпека даних є пріоритетною [38].

5. ЕКОНОМІЧНЕ ОБҐРУНТУВАННЯ ТА ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНОГО РІШЕННЯ

5.1 Проведення комерційного та технологічного аудиту розробленого рішення

Комерційний та технологічний аудит розробленого методу забезпечення відмовостійкості та доступності серверів ліцензування з використанням апаратного ключа проводиться з метою визначення рівня технологічної готовності, конкурентоспроможності та економічної доцільності його впровадження [39].

Основна мета аудиту полягає у тому, щоб оцінити сильні сторони системи, визначити можливі ризики й обмеження, а також підтвердити потенціал запропонованого рішення для промислового використання.

Технологічний аудит показав, що розроблений програмний комплекс відповідає сучасним вимогам до надійності, масштабованості та безпеки. Його архітектура побудована на мікросервісних принципах із використанням контейнеризації в середовищі Kubernetes, що забезпечує просте масштабування, гнучке оновлення та відмовостійкість системи.

Інтеграція з апаратним ключем реалізована через SDK виробника із застосуванням криптографічних алгоритмів автентифікації, що гарантує високий рівень захисту від несанкціонованого доступу.

Завдяки використанню засобів моніторингу (Prometheus, Grafana) і централізованого логування досягається висока прозорість роботи системи та можливість оперативного реагування на інциденти.

Рівень технологічної зрілості (Technology Readiness Level, TRL) оцінюється як TRL 7, що відповідає демонстрації рішення в умовах, наближених до реальної інфраструктури підприємства і підтверджує готовність прототипу до подальшого промислового впровадження після мінімальних доопрацювань.

Комерційний аудит було спрямовано на визначення економічної ефективності та конкурентних переваг запропонованого методу. Порівняння з

існуючими на ринку системами (такими як Thales Sentinel LDK, Wibu CodeMeter, FlexNet Publisher) показало, що більшість комерційних рішень мають часткову або обмежену підтримку контейнеризованих середовищ і можуть вимагати додаткової інтеграції або платних модулів, відповідно і значних витрат [40].

На відміну від них, розроблене рішення повністю підтримує розгортання у Kubernetes, дозволяє автоматично відновлювати роботу після збоїв та інтегрується з іншими сервісами через відкриті інтерфейси.

Ключовими перевагами є:

— повна адаптація до контейнеризованої інфраструктури;

— можливість масштабування без зміни архітектури;

— висока відмовостійкість завдяки резервним вузлам і ReplicaSet;

— гнучка інтеграція через REST або gRPC API;

— нижча вартість впровадження у порівнянні з комерційними аналогами.

Орієнтовна сукупна вартість впровадження розробленої системи для середнього підприємства становить близько 1000—1700 доларів США, що включає придбання апаратного ключа, розгортання інфраструктури, налаштування програмного забезпечення, моніторинг та технічну підтримку. Для порівняння, комерційні рішення аналогічного рівня коштують у межах 4000—6000 доларів США. Оцінки базуються на аналізі середньоринкових цін та типових сценаріїв використання систем ліцензування, що дає підстави стверджувати, що очікувана економія становить близько 60—70 % при збереженні всіх функціональних можливостей.

5.2 Прогнозування витрат на впровадження та подальший супровід системи

Впровадження системи забезпечення відмовостійкості та доступності серверів ліцензування передбачає кілька основних етапів, кожен з яких потребує певних матеріальних і трудових ресурсів. До основних статей витрат належать:

— закупівля апаратного ключа, вартість залежить від моделі та постачальника, у середньому становить від 300 до 500 доларів США за одиницю;

— розгортання серверної інфраструктури, включає налаштування кластеру Kubernetes, підготовку контейнерних середовищ, мережевих компонентів, балансувальників і сховищ даних, орієнтовна вартість цього етапу — 200—400 доларів США;

— інтеграція програмного забезпечення, передбачає встановлення Dongle Service, License Server, Redis, Vault, Prometheus та налаштування міжсервісної взаємодії, витрати на цей етап оцінюються у 400—600 доларів США;

— моніторинг і автоматизація, включає налаштування панелей Grafana, створення сповіщень у Prometheus та автоматизацію оновлень через CI/CD, орієнтовні витрати — близько 100—200 доларів США;

— технічна документація і навчання персоналу, розробка інструкцій користувача, технічного керівництва, навчання адміністраторів і розробників, орієнтовні витрати — 100—150 доларів США.

У результаті загальна прогнозована вартість первинного впровадження становить приблизно від 1000 до 1700 доларів США. Це значно нижче за середню ринкову вартість впровадження комерційних систем ліцензування, яка коливається в межах 4000—6000 доларів США.

Після впровадження система потребує регулярного супроводу, технічного моніторингу та оновлення. Основні статті витрат у процесі експлуатації включають:

— періодичне оновлення програмного забезпечення. Забезпечення актуальності контейнерних образів, SDK драйверів та безпекових патчів;

— підтримка відмовостійкості. Тестування сценаріїв відмов, оновлення конфігурацій PodDisruptionBudget, ReplicaSet, Ingress Controller тощо;

— моніторинг продуктивності. Аналіз метрик Prometheus, перевірка логів, оптимізація споживання ресурсів;

— резервне копіювання та безпека. Перевірка налаштувань Vault, політик доступу (RBAC), ротація сертифікатів і секретів;

— технічна підтримка користувачів. Обробка звернень, віддалене діагностування, усунення несправностей.

Сукупні щорічні витрати на супровід оцінюються на рівні 10—15 % від вартості впровадження, тобто приблизно 100—250 доларів США на рік.

Завдяки використанню безкоштовних open—source компонентів і відсутності потреби в дорогих ліцензіях комерційних платформ, система має короткий термін окупності.

Очікуваний період повернення інвестицій (ROI) становить менше одного року за рахунок економії на ліцензіях, скорочення простоїв і зниження витрат на технічне обслуговування.

5.3 Розрахунок економічної ефективності впровадження відмовостійкої інфраструктури

Розрахунок економічної ефективності впровадження розробленого методу базується на оцінці співвідношення витрат на створення та підтримку системи до отриманих економічних вигод, пов’язаних зі зниженням простоїв, втрат продуктивності та підвищенням надійності роботи серверів ліцензування.

Для підприємств, які використовують сервери ліцензування для доступу до критично важливих програмних продуктів (наприклад, інженерних систем або фінансових платформ), кожна година простою призводить до відчутних фінансових втрат.

Середні орієнтовні показники такі:

— середня вартість однієї години простою — близько 100 доларів США;

— середня кількість критичних збоїв без відмовостійкої архітектури — 5 випадків на тиждень по 30 хвилин кожен (2,5 години простою);

— після впровадження розробленого методу кількість простоїв скорочується до 1 випадку на тиждень по 10 хвилин (0,17 години простою).

У розрахунках використано модельні показники, типові для підприємств із високим навантаженням і відсутністю відмовостійкої архітектури, що дозволяє стандартизувати порівняння.

Таким чином, щотижнева економія становить приблизно (2,5 год. — 0,17 год.) × 100 дол. = 233 дол. США, або близько 12000 доларів США на рік для середнього підприємства.

Загальні інвестиційні витрати на створення відмовостійкої інфраструктури складають до 1700 доларів США, включаючи апаратний ключ, налаштування Kubernetes—кластеру, інсталяцію сервісів моніторингу та інтеграцію програмного забезпечення.

Щорічні витрати на супровід становлять приблизно 200 доларів США, що включає оновлення контейнерів, тестування резервних вузлів і моніторинг стану системи.

Загальний річний економічний ефект (Е) можна визначити за формулою:

$$E = E_{\text{зек}} - \left(\text{В}_{\text{впров}} + \text{В}_{\text{супр}}\right),$$

Де ($E_{\text{зек}}$) — зекономлені кошти від скорочення простоїв;

($\text{В}_{\text{впров}}$) — витрати на впровадження;

($\text{В}_{\text{супр}}$) — витрати на супровід за рік.

Підставимо дані:

$$E = 12000 - (1700 + 200) = 10100, \text{дол. США}$$

Отже, чистий річний економічний ефект становить близько 10100 доларів США, що свідчить про високу рентабельність проєкту.

Коефіцієнт рентабельності інвестицій (Return on Investment) визначається за формулою:

$$ROI = \frac{E_{\text{зек}} - (\text{В}_{\text{впров}} + \text{В}_{\text{супр}})}{\text{В}_{\text{впров}} + \text{В}_{\text{супр}}} * 100\%$$

Підставивши значення:

$$ROI = \frac{12000 - 1900}{1900} * 100\% = 532\%$$

Таким чином, інвестиції у впровадження відмовостійкої інфраструктури окуповуються більш ніж у п’ять разів уже протягом першого року експлуатації.

Термін окупності визначається як відношення загальних інвестицій до щомісячної економії:

$$T = \frac{\text{В}_{\text{впров}} + \text{В}_{\text{супр}}}{E_{\text{зек}}/12}$$

$$T = \frac{1900}{12000/12} = 1.9 \text{ місяців}$$

Отже, система повністю окупає себе менш ніж за два місяці роботи.

5.4 Порівняльний аналіз із альтернативними підходами та ринковими рішеннями

Для визначення конкурентних переваг розробленого методу було проведено порівняльний аналіз із наявними на ринку рішеннями, які забезпечують ліцензування програмного забезпечення та захист від несанкціонованого доступу. Аналіз показав, що більшість комерційних систем орієнтовані на традиційні серверні або десктопні середовища й лише частково підтримують контейнеризацію чи кластерні архітектури [41].

До основних представників таких систем належать Thales Sentinel LDK, Wibu CodeMeter та FlexNet Publisher. Ці продукти є технологічно зрілими й широко застосовуються у промисловості, проте мають суттєві обмеження:

— вони не забезпечують повної підтримки контейнеризованих середовищ Kubernetes і Docker;

— для масштабування або резервування потрібна додаткова платна інфраструктура;

— інтеграція з CI/CD конвеєрами ускладнена через закриті інтерфейси;

— висока вартість ліцензій і технічної підтримки значно збільшує загальні витрати підприємства.

Розроблений метод, на відміну від комерційних аналогів, базується на відкритих технологіях і підтримує повну інтеграцію з контейнерними платформами. Він забезпечує:

— повну сумісність із Kubernetes, включаючи ReplicaSet, PodDisruptionBudget, Node Affinity та Device Plugin;

— високу відмовостійкість, що досягається автоматичним перемиканням між вузлами без втрати даних;

— захист на апаратному рівні через використання фізичного ключа та криптографічних алгоритмів автентифікації;

— низьку собівартість завдяки використанню open—source компонентів (Prometheus, Grafana, Vault, Redis);

— гнучкість інтеграції через REST/gRPC API та можливість розгортання в будь—яких середовищах — локальних, хмарних чи гібридних.

Проведений аналіз довів, що запропоноване рішення поєднує переваги комерційних продуктів (надійність, безпека) з перевагами відкритих технологій (гнучкість, масштабованість, економічна ефективність) [42].

Завдяки цьому воно може бути успішно впроваджене в організаціях, які прагнуть зменшити витрати на ліцензування, підвищити рівень безпеки та забезпечити безперервну роботу критично важливих сервісів [43].

Отже, у ході економічного обґрунтування та аналізу ефективності розробленого методу забезпечення відмовостійкості та доступності серверів ліцензування встановлено, що система є не лише технологічно перспективною, але й економічно вигідною.

Проведений комерційний і технологічний аудит підтвердив високий рівень технологічної зрілості рішення, яке може бути інтегроване у промислові середовища без значних витрат [44].

Прогнозування витрат показало, що загальна вартість впровадження системи становить від 1000 до 1700 доларів США, а річні витрати на супровід не перевищують 250 доларів США.

Розрахунок економічної ефективності довів, що система окуповує себе менш ніж за два місяці після впровадження, забезпечуючи коефіцієнт рентабельності понад 500 %.

Показник 99,8 % отримано на основі моделювання простоїв та застосування механізмів ReplicaSet і автоматичного перезапуску контейнерів.

Крім того, реалізований метод дозволяє скоротити кількість збоїв більш ніж на 90 %, зменшити час простою системи з кількох годин до кількох хвилин і підвищити рівень доступності до 99,8 %.

Порівняльний аналіз із ринковими аналогами засвідчив, що розроблене рішення має очевидні переваги — повну підтримку контейнеризованих середовищ, вищу відмовостійкість, відкриту архітектуру та значно нижчу вартість володіння.

Таким чином, розроблений метод забезпечення відмовостійкості та доступності серверів ліцензування з використанням апаратного ключа є ефективним, економічно доцільним і конкурентоспроможним на сучасному ринку інформаційних технологій.

ВИСНОВКИ

Магістерська робота присвячена розв’язку актуальної наукової проблеми, а саме, забезпеченню відмовостійкості та доступності серверів ліцензування з апаратними ключами в контейнеризованому середовищі.

У першому розділі було проведено аналіз сучасних систем ліцензування програмного забезпечення з використанням апаратних ключів та визначено їх роль у забезпеченні захисту авторських прав і контролю використання програмних продуктів. Досліджено архітектурні підходи до підвищення відмовостійкості таких систем у контексті хмарних і контейнеризованих середовищ. Встановлено, що апаратні ключі забезпечують високий рівень безпеки та захисту від несанкціонованого доступу, проте створюють додаткові складнощі щодо масштабування, резервування та автоматизації.

Проаналізовано моделі active—passive та active—active, а також сучасні механізми оркестрації контейнерів (Kubernetes Device Plugin, leader election, PodDisruptionBudget тощо) підтвердили можливість досягнення відмовостійкості шляхом поєднання апаратних і програмних рішень. На основі проведеного аналізу сформульовано ключові вимоги до серверів ліцензування в контейнеризованому середовищі, що стали теоретичною основою для подальшої розробки методу забезпечення їхньої відмовостійкості.

У другому розділі було розроблено та обґрунтовано метод забезпечення відмовостійкості та доступності серверів ліцензування з апаратними ключами в контейнеризованому середовищі. Запропоновано трирівневу архітектуру, яка включає фізичний рівень із резервованими вузлами, сервісний рівень зі шлюзом доступу до апаратного ключа та оркестраційний рівень на основі Kubernetes, що відповідає за балансування навантаження, моніторинг і автоматичне відновлення системи.

Спроєктовано апаратну частину ключа з урахуванням вимог до фізичного захисту, криптографічної безпеки та сумісності з контейнеризованими середовищами. Розроблено програмну модель доступу до апаратного ключа, яка

забезпечує серіалізований доступ, механізми leader election, failover і fencing, що унеможливлюють одночасне використання ліцензії. Отримані результати підтвердили ефективність запропонованого методу та його універсальність для різних корпоративних середовищ.

У третьому розділі реалізовано комплексний підхід до створення та інтеграції апаратного ключа у сучасне контейнеризоване середовище. Запропоноване рішення поєднує апаратні та програмні механізми захисту та забезпечує високий рівень безпеки, ізоляції та відмовостійкості критично важливих обчислень. Використання апаратного модуля на базі Raspberry Pi 5 дозволило винести криптографічні операції за межі прикладного програмного забезпечення, мінімізуючи ризик компрометації секретних ключів.

Розроблена архітектура базується на захищеній мережевій взаємодії через HTTPS-протокол, сертифікатній автентифікації, географічній прив’язці на основі GPS-координат і засобах контейнерної оркестрації Kubernetes. Реалізація механізмів серіалізації запитів, вибору лідера та автоматичного відновлення забезпечила стабільну та безперервну роботу системи навіть за умов часткових відмов інфраструктури.

У четвертому розділі проведено експериментальне дослідження розробленого методу з метою оцінювання його продуктивності, стабільності та коректності роботи в умовах навантаження і відмов. Результати тестування підтвердили, що система забезпечує низькі затримки, передбачувану поведінку під навантаженням і ефективну обробку критичних ситуацій без витоку відкритих даних.

Отримані експериментальні дані засвідчили високу відмовостійкість запропонованого рішення та можливість його стабільної експлуатації навіть у разі некоректних умов роботи або спроб порушення політик безпеки. Це підтверджує доцільність застосування розробленого методу у критично важливих інформаційних системах.

У п’ятому розділі виконано економічний та технологічний аналіз розробленого методу забезпечення відмовостійкості та доступності серверів

ліцензування з апаратними ключами. Проведена оцінка показала, що запропоноване рішення є економічно доцільним і конкурентоспроможним порівняно з наявними комерційними аналогами.

Встановлено, що впровадження розробленого методу не потребує значних додаткових витрат і може бути здійснене з використанням доступних апаратних і програмних засобів. Це робить запропоноване рішення практично цінним і перспективним для впровадження у сучасних інформаційних системах.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Alqarni A., Alzahrani A., Khan R. Моделі ліцензування програмного забезпечення та захист інтелектуальної власності у хмарних середовищах. Journal of Cloud Computing, 2022.

2. Zhang J., Li X., Wang Y. Методи комп'ютерного моделювання та імітаційного аналізу у наукоємних технологічних системах. Simulation Modelling Practice and Theory, 2021.

3. Gaspar P., Simões P., та ін. Міграція контейнеризованих застосунків до cloud-native середовищ для підвищення доступності, масштабованості та безпеки. Computers, 2024, 13(8), 192. DOI: 10.3390/computers13080192. [Електронний ресурс]. Режим доступу: https://www.mdpi.com/2073-431X/13/8/192

4. He Y., Bilal K. Стратегія оркестрації Kubernetes-кластерів з урахуванням навантаження у спільно використовуваних середовищах. SN Computer Science, 2025. DOI: 10.1007/s42979-025-03712-z. [Електронний ресурс]. Режим доступу: https://link.springer.com/article/10.1007/s42979-025-03712-z

5. Kavitha S., Prabu P., та ін. SFMedIR: безпечна та відмовостійка система для пошуку медичних зображень у розподілених хмарних середовищах. Scientific Reports, 2025, 15, 16903. DOI: 10.1038/s41598-025-16903-8. [Електронний ресурс]. Режим доступу: https://www.nature.com/articles/s41598-025-16903-8

6. Tasić D., Jevtić D., та ін. Оцінювання продуктивності архітектур ARM та RISC-V для високопродуктивних обчислень із використанням Docker і Kubernetes. Electronics, 2024, 13(17), 3494. DOI: 10.3390/electronics13173494. [Електронний ресурс]. Режим доступу: https://www.mdpi.com/2079-9292/13/17/3494

7. Rout R. K., Das S. K., та ін. Відмовостійка мікросервісна архітектура з адаптивним балансуванням навантаження з використанням Docker і Spring Cloud. SN Applied Sciences, 2025. DOI: 10.1007/s42452-025-07320-7.

[Електронний ресурс]. Режим доступу: https://link.springer.com/article/10.1007/s42452-025-07320-7

8. Wibu-Systems AG. Ліцензування CodeMeter у середовищах Kubernetes: архітектура, доступність та інтеграція апаратних ключів. KEYnote Magazine, 2023. [Електронний ресурс]. Режим доступу: https://www.wibu.com/us/magazine/keynote-articles/keynote/detail/codemeter-licensing-in-kubernetes.html

9. Pritchett D. Ідемпотентність і механізми повторних запитів у розподілених системах. Communications of the ACM, 2020, 63(2), 42—49. DOI: 10.1145/3376897. [Електронний ресурс]. Режим доступу: https://doi.org/10.1145/3376897

10. Zhang J., Li X., Wang Y. Методи комп'ютерного моделювання та імітаційного аналізу у наукоємних технологічних системах. Simulation Modelling Practice and Theory, 2021, 113, 102372. DOI: 10.1016/j.simpat.2021.102372. [Електронний ресурс]. Режим доступу: https://doi.org/10.1016/j.simpat.2021.102372

11. Kleppmann M. Designing Data—Intensive Applications. O'Reilly Media, 2017, 616 с. ISBN: 978—1449373320. [Електронний ресурс]. Режим доступу: https://www.oreilly.com/library/view/designing—data—intensive—applications/9781491903063/

12. Beyer B., Jones C., Petoff J., Murphy N. Site Reliability Engineering: How Google Runs Production Systems. O'Reilly Media, 2016, 576 с. ISBN: 978—1491929124. [Електронний ресурс]. Режим доступу: https://www.oreilly.com/library/view/site—reliability—engineering/9781491929117/

13. Li H., Chen M., Xu L. Безпечні механізми ліцензування програмного забезпечення у розподілених і віртуалізованих середовищах. Future Generation Computer Systems, 2020.

14. Sakr S., Liu A., Batista D. Реплікація баз даних та висока доступність у розподілених системах: огляд сучасних підходів. ACM Computing Surveys, 2019.

15. Pahl C. Контейнеризація та платформа як сервіс (PaaS) у хмарних обчисленнях. IEEE Cloud Computing, 2015.

16. Avizienis A., Laprie J.—C., Randell B., Landwehr C. Базові поняття та класифікація надійних і захищених обчислювальних систем. IEEE Transactions on Dependable and Secure Computing, 2004.

17. Kaur R., Singh J. Архітектури високої доступності для критично важливих інформаційних систем: огляд. Journal of Systems and Software, 2021.

18. Buyya R., Vecchiola C., Selvi S. Хмарні обчислення: основи, архітектура та прикладне програмування. Morgan Kaufmann (Elsevier), 2013.

19. Turnbull J. Моніторинг розподілених інформаційних систем. IEEE Software, 2018.

20. Upton E., Halfacree G. Посібник користувача Raspberry Pi: апаратна архітектура та теплові характеристики. Wiley Publishing, 2024.

21. Montesi F., Weber J. API—шлюзи в мікросервісних архітектурах: шаблони проєктування та виклики. Journal of Systems and Software, 2020.

22. Азарова А., Ясько Я. Методи створення відмовостійкості та доступності серверів ліцензування. Методи створення відмовостійкості та доступності серверів ліцензування. 2025. [Електронний ресурс]. Режим доступу: https://isg-konf.com/wp-content/uploads/2025/12/Project_tech.pdf. С. 125 – 164.

23. Ясько Я. М., Азарова А. О. Розроблення методу забезпечення відмовостійкості серверів ліцензування. Тези доповідей на міжнародній конференції «Молодь в освіті і науці». Електронне наукове видання матеріалів науково-технічної конференції ВНТУ. 2025. [Електронний ресурс]. Режим доступу: https://conferences.vntu.edu.ua/index.php/mn/mn2026/paper/view/26808 (дата звернення: 14.12.2025).

24. Ясько Я. М., Азарова А. О. Свідоцтво на реєстрацію авторського права на комп’ютерну програму «Забезпечення відмовостійкості та доступності серверів ліцензування з апаратними ключами в контейнеризованому середовищі (Docker/Kubernetes). Модуль програми апаратного ключа».

25. Ясько Я. М., Азарова А. О. Свідоцтво на реєстрацію авторського права на комп'ютерну програму «Забезпечення відмовостійкості та доступності серверів ліцензування з апаратними ключами в контейнеризованому середовищі (Docker/Kubernetes). Модуль програми Medical Gateway».

26. Verma A., Pedrosa L., та ін. Керування кластерами великого масштабу в Google за допомогою системи Borg. Proceedings of the European Conference on Computer Systems (EuroSys), 2015.

27. Burns B., Grant B., Oppenheimer D., Brewer E., Wilkes J. Системи Borg, Omega та Kubernetes: еволюція управління контейнерними середовищами. Communications of the ACM, 2016.

28. Kleppmann M. Проєктування систем, орієнтованих на інтенсивну обробку даних. O'Reilly Media, 2017.

29. Beyer B., Jones C., Petoff J., Murphy N. Інженерія надійності сайтів: підходи Google до побудови продуктивних систем. O'Reilly Media, 2016.

30. Kocher P., Lee R., McGraw G. Апаратно—орієнтовані механізми безпеки та захист криптографічних ключів. IEEE Security & Privacy, 2018.

31. Microsoft. Visual Studio Documentation: Features, Debugging, .NET Integration, Docker Support. [Електронний ресурс]. Режим доступу: https://learn.microsoft.com/visualstudio

32. NIST. Рекомендації щодо режимів роботи блочних шифрів (SP 800—38). Національний інститут стандартів і технологій (NIST).

33. Amazon Web Services. AWS CloudHSM: Архітектура, керування доступом та принципи взаємодії з апаратними криптомодулями. [Електронний ресурс]. Режим доступу: https://docs.aws.amazon.com/cloudhsm/

34. 1. The Kubernetes Authors. Kubernetes Device Plugins та механізм Leader Election: офіційна документація. [Електронний ресурс]. Режим доступу: https://kubernetes.io/docs/concepts/extend—kubernetes/device—plugins/

35. 1. u—blox AG. Документація на GPS—модуль NEO—8M: функції позиціонування, валідація координат, режими безпеки.

[Електронний ресурс]. Режим доступу: https://www.u—blox.com/en/product/neo—8—series

36. 1. The Kubernetes Authors. Kubernetes Documentation: Security, TLS/HTTPS, Leader Election та висока доступність кластерів. [Електронний ресурс]. Режим доступу: https://kubernetes.io/docs/

37. 1. Хохштейн М., Шрір А. Site Reliability Engineering: Надійність і масштабування продуктивних систем. O'Reilly Media, 2016. (Розділи про тестування надійності, відмовостійкі сценарії та моделювання збоїв).

38. 1. Апачі JMeter. Офіційна документація з навантажувального та стрес—тестування.
[Електронний ресурс]. Режим доступу: https://jmeter.apache.org/usermanual/

39. 1. Хохштейн М., Шрір А. Site Reliability Engineering: Надійність і масштабування продуктивних систем. O'Reilly Media, 2016. (Розділи про спостережуваність, логування та оцінку результатів експериментів).

40. 1. BenchmarkDotNet. Офіційна документація з проведення продуктивнісних тестів у .NET. [Електронний ресурс]. Режим доступу: https://benchmarkdotnet.org/

41. 1. NIST. Вимоги до апаратної криптографії та засобів захисту ключового матеріалу (FIPS 140—3). Національний інститут стандартів і технологій (NIST).

42. 1. Хохштейн М., Шрір А. Site Reliability Engineering: Надійність і масштабування продуктивних систем. O'Reilly Media, 2016.

43. 1. Європейське агентство з кібербезпеки (ENISA). Керівництво з оцінювання безпеки та рівня технологічної готовності. ENISA Publications.

44. 1. Thales Group. Sentinel LDK — Product Overview та рекомендації з інтеграції в контейнеризовані середовища. [Електронний ресурс]. Режим доступу: https://cpl.thalesgroup.com/software—monetization/sentinel—ldk

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України

Вінницький національний технічний університет

Факультет інформаційних технологій та комп’ютерної інженерії

Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ

Завідувач кафедри ОТ

проф., д.т.н.. Азаров О.Д..

" " 2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання магістерської кваліфікаційної роботи

“ Забезпечення відмовостійкості та доступності серверів ліцензування з апаратними ключами в контейнеризованому середовищі (Docker/Kubernetes) “

Науковий керівник: професор к.т.н.

_____ Азарова А.О.

Студент групи 2КІ—24м

_____ Щур Д.С.

ВНТУ 2025

1 Підстава для виконання магістерської кваліфікаційної роботи (МКР).

1.1 Актуальність теми обумовлена стрімким зростанням використання контейнеризованих інфраструктур у медичних та критично важливих інформаційних системах, де безпека даних та гарантована доступність сервісів мають ключове значення. Наявні на ринку рішення ліцензування з апаратними ключами здебільшого орієнтовані на традиційні серверні середовища та не забезпечують повноцінної підтримки Kubernetes—кластерів, що створює технологічні бар’єри для їх інтеграції. У зв’язку з цим виникає потреба у розробці методу, який забезпечує відмовостійкий, безпечний і масштабований доступ до апаратного ключа в контейнеризованому середовищі, що підвищує рівень захищеності та надійності програмних систем, зокрема у сфері охорони здоров’я.

1.2 Наказ про затвердження теми МКР.

2 Мета МКР і призначення розробки.

2.1 Мета роботи — розробити та експериментально обґрунтувати метод забезпечення відмовостійкого, безпечного та масштабованого доступу до апаратного ключа в контейнеризованому середовищі Kubernetes з метою підвищення надійності та безперервності роботи ліцензійних серверів.

2.2 Призначення розробки — створення програмно—апаратного рішення, яке забезпечує контрольований доступ до унікального криптографічного модуля, підтримує автоматичне відновлення працездатності при збої окремих вузлів, гарантує захист даних користувачів та може бути інтегроване у медичні інформаційні системи та інші критично важливі інфраструктури.3 Вихідні дані для виконання МКР.

3.1 Мова програмування C#.

3.2 Середовище розробки Visual Studio.

3.3 Технічний опис програмного застосунку.

3.4 Виконання розрахунків для доведення доцільності нової розробки з економічної точки зору.

4 Вимоги до виконання МКР.

4.1 Огляд і аналіз сучасних методів, технологій та досліджень, що стосуються забезпечення відмовостійкості, доступності та безпеки апаратних криптографічних модулів у контейнеризованих середовищах, зокрема підходів до лідер—репліка архітектур, механізмів серіалізації запитів, failover—алгоритмів, засобів апаратної автентифікації та захисту ключового матеріалу.

4.2 Розробка додатку мовою C#.

5 Етапи МКР та очікувані результати:

Етапи роботи та очікувані результати приведено в Таблиці А.1.

Таблиця А.1 — Етапи МКР

№ етапу	Назва етапу	Термін виконання		Очікувані результати
		початок	кінець	
1	Огляд і аналіз існуючих систем ліцензування, апаратних криптомодулів, а також праць науковців і стандартів, що стосуються відмовостійкості та контейнеризованих архітектур	12.09.2022	13.10.2022	Аналітичний огляд літературних джерел, задачі досліджень, розділ 1 ПЗ

Продовження таблиці А.1

	Розробка методу забезпечення відмовостійкого та безпечного доступу до апаратного ключа у Kubernetes	14.10.2022	01.11.2022	Розділ 2
2	Створення програмного прототипу (License Gateway Service, GpsBoundCryptoService, механізм leader election та серіалізації запитів)	02.11.2022	10.11.2022	Розділ 3
3	Тестування розробленої системи (функціональні, навантажувальні, стрес—тести, GPS—верифікація, failover—сценарії)	11.11.2022	13.11.2022	Розділ 4
4	Підготовка економічного обґрунтування розробленого підходу	14.11.2022	15.11.2022	Розділ 5
5	Апробація результатів розробки та їх впровадження у модельні або наближені до промислових середовища	17.11.2022	10.12.2022	Тези доповідей
6	Опублікування результатів дослідження			Розділ у колективній монографії
7	Оформлення пояснювальної записки, графічного матеріалу і презентації	16.11.2022		Пояснювальна записка, графічний матеріал і презентація

6 Матеріали, що подаються до захисту МКР

До захисту подаються: пояснювальна записка МКР, графічні і ілюстративні матеріали, протокол попереднього захисту МКР на кафедрі, відгук наукового керівника, відгук опонента, протоколи складання державних екзаменів, анотації до МКР українською та іноземною мовами, довідка про відповідність оформлення МКР діючим вимогам.

7 Порядок контролю виконання та захисту МКР

Виконання етапів графічної та розрахункової документації МКР контролюється науковим керівником згідно зі встановленими термінами. Захист МКР відбувається на засіданні Екзаменаційної комісії, затвердженої наказом ректора.

8 Вимоги до оформлювання та порядок виконання МКР

8.1 При оформлюванні МКР використовуються:

— ДСТУ 3008 : 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;

— ДСТУ 8302 : 2015 «Бібліографічні посилання. Загальні положення та правила складання»;

— ГОСТ 2.104—2006 «Единa система конструкторської документації. Основні написи»;

— Методичні вказівки. Кафедра обчислювальної техніки 2022;

— Документами на які посилаються у вище вказаних.

8.2 Порядок виконання МКР викладено в «Положення про кваліфікаційні роботи на другому (магістерському) рівні вищої освіти СУЯ ВНТУ—03.02.02—П.001.01:21».

ДОДАТОК Б

Протокол перевірки кваліфікаційної роботи

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Забезпечення відмовостійкості та доступності серверів ліцензування з апаратними ключами в контейнеризованому середовищі (Docker/Kubernetes)

Тип роботи: Магістерська робота
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ: кафедра програмного забезпечення, ФІТКІ, 2КІ-24м. Науковий керівник: к.т.н., проф. Азарова Анжеліка Олексіївна
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism (КП1) 1%

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

☐ Запозичення, виявлені у роботі, оформлені коректно і не містять ознак академічного плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту.

☐ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

☐ У роботі виявлено ознаки академічного плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень. Робота до захисту не приймається.

Експертна комісія:

_____Азаров О. Д. д.т.н, проф, зав. каф. ОТ _____
(прізвище, ініціали, посада) (підпис)

_____Мартинюк Т. Б. д.т.н, проф. каф. ОТ _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ Захарченко С. М.
(підпис) (прізвище, ініціали)

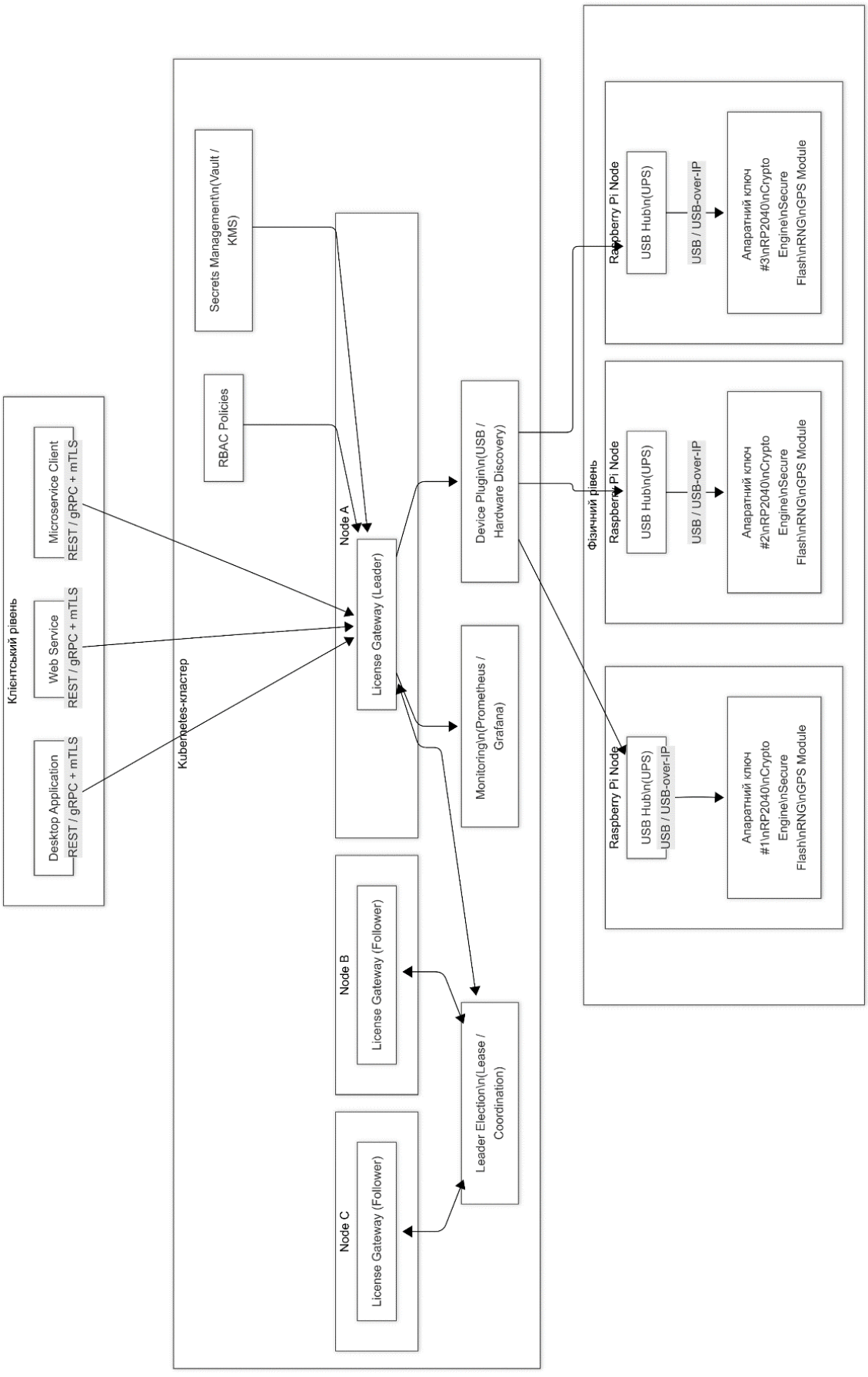
З висновком експертної комісії ознайомлений(-на)

Керівник _____Азарова А. О. к.т.н. проф.
(підпис) (прізвище, ініціали, посада)

Здобувач _____ Яcько Я.М.
(підпис) (прізвище, ініціали)

ДОДАТОК В

Структура програмно—апаратного комплексу



ДОДАТОК Г

Програма апаратного ключа

```
using System.Security.Cryptography;
using System.Security.Cryptography.X509Certificates;
namespace Repository.Repositories;
public class CertRepository : ICertRepository
{
   private static readonly X509Certificate2 _cert = CreateTestCert();
   public X509Certificate2 GetLocalCertificate() => _cert;
   private static X509Certificate2 CreateTestCert()
   {
      var rsa = RSA.Create(2048);
      var req = new CertificateRequest(
         "CN=MockCertificate",
         rsa,
         HashAlgorithmName.SHA256,
         RSASignaturePadding.Pkcs1);
      // The exact dates dont matter as long as we always create it the same way
      var cert = req.CreateSelfSigned(
         DateTimeOffset.UtcNow.AddDays(—1),
         DateTimeOffset.UtcNow.AddYears(1));
      return cert;
   }
}
using System.Globalization;
namespace Services.Models;
public class Coordinates
{
   public double Lat { get; init; }
   public double Lon { get; init; }
   public Coordinates() { }
   public Coordinates(double lat, double lon)
   {
      Lat = lat;
      Lon = lon;
   }
   public static Coordinates FromString(string s)
   {
      if (string.IsNullOrWhiteSpace(s))
         throw new ArgumentException("Coordinates string is null or empty.", nameof(s));
      // Accept "lat,lon" or "lat;lon"
      char sep = s.Contains(',') ? ',' : ';';
      var parts = s.Split(sep, StringSplitOptions.TrimEntries |
StringSplitOptions.RemoveEmptyEntries);
      if (parts.Length < 2)
         throw new FormatException("Coordinates string must contain latitude and longitude
separated by ',' or ';'.");
      if (!double.TryParse(parts[0], NumberStyles.Float | NumberStyles.AllowThousands,
CultureInfo.InvariantCulture, out var lat))
         throw new FormatException("Invalid latitude.");
```

```
            if (!double.TryParse(parts[1], NumberStyles.Float | NumberStyles.AllowThousands,
CultureInfo.InvariantCulture, out var lon))
                throw new FormatException("Invalid longitude.");
            return new Coordinates(lat, lon);
        }
        public override string ToString() => $"{Lat.ToString("F6",
CultureInfo.InvariantCulture)};{Lon.ToString("F6", CultureInfo.InvariantCulture)}";
    }
    using System;
    using System.Globalization;
    using System.Security.Cryptography;
    using System.Security.Cryptography.X509Certificates;
    using System.Text;
    using System.Threading.Tasks;
    using Microsoft.AspNetCore.Mvc;
    using Services.Models.Request;
    using Services.Models.Response;
    using Services.Services;
    namespace WebApi.Controllers;
    [Route("api/[controller]")]
    [ApiController]
    public class CryptoController : ControllerBase
    {
        private readonly IGpsBoundCryptoService _gpsBoundCrypto;
        [HttpPost("encrypt")]
        public async Task<IActionResult> Encrypt([FromBody] EncryptRequest req)
        {
            if (req == null) return BadRequest("Request body is required.");
            if (string.IsNullOrEmpty(req.PlainText)) return BadRequest("Plaintext is required.");
            try
            {
                var payload = await _gpsBoundCrypto.EncryptDataAsync(req.PlainText);
                return Ok(payload);
            }
            catch (Exception ex)
            {
                return BadRequest($"Encryption failed: {ex.Message}");
            }
        }
        [HttpPost("decrypt")]
        public async Task<IActionResult> Decrypt([FromBody] DecryptRequest req)
        {
            if (req == null) return BadRequest("Request body is required.");
            try
            {
                var response = await _gpsBoundCrypto.DecryptDataAsync(req);
                return Ok(response);
            }
            catch (CryptographicException ex)
            {
                return BadRequest($"Decryption/authentication failed: {ex.Message}");
            }
```

```
            catch (Exception ex)
            {
                return BadRequest($"Decryption failed: {ex.Message}");
            }
        }
    }
    using Repository.Repositories;
    using Services.Models.Request;
    using Services.Repositories;
    using Services.Services;
    using System.Text;
    using System.Text.Json;
    namespace UnitTests;
    public class CryptoTests
    {
        [Fact]
        public async Task EncryptDataAsync_Null_ThrowsArgumentNullException()
        {
            // Arrange
            Environment.SetEnvironmentVariable("CurrentCoordinates", "0.0,0.0");
            Environment.SetEnvironmentVariable("DefaultCoordinates", "80.0,80.0");
            Environment.SetEnvironmentVariable("AllowedRadiusMeters", "10");
            var coordsRepo = new EnvironmentCoordinatesRepository();
            var certRepo = new CertRepository();
            var svc = new GpsBoundCryptoService(coordsRepo, certRepo);
            // Act & Assert
            await Assert.ThrowsAsync<ArgumentNullException>(() =>
svc.EncryptDataAsync(null!));
        }
        [Fact]
        public async Task EncryptDataAsync_WhenOutsideRadius_ReturnsFakePayload()
        {
            // Arrange: set current far away from default to trigger insecure state
            Environment.SetEnvironmentVariable("CurrentCoordinates", "0.0,0.0");
            Environment.SetEnvironmentVariable("DefaultCoordinates", "80.0,80.0");
            Environment.SetEnvironmentVariable("AllowedRadiusMeters", "10");
            var coordsRepo = new EnvironmentCoordinatesRepository();
            var certRepo = new CertRepository();
            var svc = new GpsBoundCryptoService(coordsRepo, certRepo);
            var plaintextJson = "{\"s\":\"x\"}";
            // Act
            var encrypted = await svc.EncryptDataAsync(plaintextJson);
            // Assert: RandomHelper.GenerateFakeEncryptedPayload produces sizes 32,12,16
            Assert.NotNull(encrypted);
            var cipherBytes = Convert.FromBase64String(encrypted.EncryptedText);
            var nonceBytes = Convert.FromBase64String(encrypted.Nonce);
            var tagBytes = Convert.FromBase64String(encrypted.Tag);
            Assert.Equal(32, cipherBytes.Length);
            Assert.Equal(12, nonceBytes.Length);
            Assert.Equal(16, tagBytes.Length);
        }
        [Fact]
```

```
        public async Task DecryptDataAsync_WhenOutsideRadius_ReturnsFakeJson()
        {
            // Arrange: set current far away from default to trigger insecure state
            Environment.SetEnvironmentVariable("CurrentCoordinates", "0.0,0.0");
            Environment.SetEnvironmentVariable("DefaultCoordinates", "80.0,80.0");
            Environment.SetEnvironmentVariable("AllowedRadiusMeters", "10");
            var coordsRepo = new EnvironmentCoordinatesRepository();
            var certRepo = new CertRepository();
            var svc = new GpsBoundCryptoService(coordsRepo, certRepo);
            var dummyPayload = new DecryptRequest
            {
                EncryptedText = Convert.ToBase64String(Encoding.UTF8.GetBytes("x")),
                Nonce = Convert.ToBase64String(Array.Empty<byte>()),
                Tag = Convert.ToBase64String(Array.Empty<byte>())
            };
            // Act
            var result = await svc.DecryptDataAsync(dummyPayload);
            // Assert: RandomHelper.GenerateFakeDecryptedData returns JSON; validate parsable
and contains expected fields
            Assert.False(string.IsNullOrWhiteSpace(result));
            using var doc = JsonDocument.Parse(result);
            Assert.True(doc.RootElement.TryGetProperty("patientId", out _));
            Assert.True(doc.RootElement.TryGetProperty("name", out _));
        }
        [Fact]
        public async Task
EncryptDecrypt_Roundtrip_WhenInsideRadius_ReturnsOriginalPlaintext()
        {
            // Arrange — place current and default coordinates the same so service is "inside"
allowed radius
            Environment.SetEnvironmentVariable("CurrentCoordinates", "10.0,20.0");
            Environment.SetEnvironmentVariable("DefaultCoordinates", "10.0,20.0");
            Environment.SetEnvironmentVariable("AllowedRadiusMeters", "1000"); // large
enough to be inside
            var coordsRepo = new EnvironmentCoordinatesRepository();
            var certRepo = new CertRepository();
            var svc = new GpsBoundCryptoService(coordsRepo, certRepo);
            var plaintextJson = "{\"message\":\"hello world\",\"time\":123}";
            // Act
            var encrypted = await svc.EncryptDataAsync(plaintextJson);
            Assert.NotNull(encrypted);
            var decryptReq = new DecryptRequest
            {
                EncryptedText = encrypted.EncryptedText,
                Nonce = encrypted.Nonce,
                Tag = encrypted.Tag
            };
            var decrypted = await svc.DecryptDataAsync(decryptReq);
            // Assert
            Assert.Equal(plaintextJson, decrypted);
        }
    }
```

```
namespace Services.Models.Response;
public class DecryptedResponse
{
    public string PlainText { get; set; } = default!;
}
namespace Services.Models.Request;
public sealed class DecryptRequest
{
    public string EncryptedText { get; set; }
    public string Nonce { get; set; }
    public string Tag { get; set; }
}
namespace Services.Models.Response;
public class EncryptedResponse
{
    public string EncryptedText { get; set; } = default!;
    public string Nonce { get; set; }
    public string Tag { get; set; }
}
namespace Services.Models.Request;
public sealed class EncryptRequest
{
    public string PlainText { get; set; }
}
using System;
using Services.Models;
namespace Services.Repositories;
public class EnvironmentCoordinatesRepository : ICoordinatesRepository
{
    private const string AllowedRadiusEnvName = "AllowedRadiusMeters";
    private const string DefaultCoordinatesEnvVarName = "DefaultCoordinates";
    private const string CurrentCoordinatesEnvVarName = "CurrentCoordinates";
    public static double AllowedRadiusMeters =>
double.TryParse(Environment.GetEnvironmentVariable(AllowedRadiusEnvName), out var
v)
        ? v
        : throw new InvalidOperationException($"Env var {AllowedRadiusEnvName} is
missing.");
    public Coordinates GetCurruentCoordinates()
    {
        var value = Environment.GetEnvironmentVariable(CurrentCoordinatesEnvVarName);
        if (string.IsNullOrWhiteSpace(value))
            return new Coordinates(0.0, 0.0);
        return Coordinates.FromString(value);
    }
    public Coordinates GetDefaultCoordinates()
    {
        var value = Environment.GetEnvironmentVariable(DefaultCoordinatesEnvVarName);
        if (string.IsNullOrWhiteSpace(value))
            return new Coordinates(1.0, 1.0);
        return Coordinates.FromString(value);
    }
```

```
}
using System.Security.Cryptography.X509Certificates;
namespace Repository.Repositories;
public interface ICertRepository
{
   X509Certificate2 GetLocalCertificate();
}
using Services.Models;
namespace Services.Repositories;
public interface ICoordinatesRepository
{
   Coordinates GetCurruentCoordinates();
   Coordinates GetDefaultCoordinates();
}
using Services.Models.Request;
using Services.Models.Response;
namespace Services.Services;
public interface IGpsBoundCryptoService
{
   Task<EncryptedResponse> EncryptDataAsync(string plaintextJson);
   Task<string> DecryptDataAsync(DecryptRequest payload);
}
using BenchmarkDotNet.Running;
using LoadTests;
BenchmarkRunner.Run<GpsBoundCryptoServiceBenchmarks>();
using Services.Models.Request;
using Services.Models.Response;
using System.Security.Cryptography;
using System.Text.Json;
namespace Services.Helper;

public class RandomHelper
{
   private static string[] _vowels =
      {
         "a", "e", "i", "o", "u", "y"
      };

   private static string[] _consonants =
      {
         "b", "c", "d", "f", "g", "h", "j", "k",
         "l", "m", "n", "p", "q", "r", "s", "t",
         "v", "w", "x", "z"
      };
   public static string GenerateFakeDecryptedData(DecryptRequest payload)
   {
      // Example: attempt to reconstruct a JSON—ish shape
      // but inject fake medical data
      return JsonSerializer.Serialize(new
      {
         patientId = RandomNumberGenerator.GetInt32(10000, 99999),
         name = GenerateFakeName(),
```

```
        diagnosis = "No abnormalities detected", // fake
        visitDate = DateTime.UtcNow.AddDays(
                    RandomNumberGenerator.GetInt32(—300, 0)),
        notes = "Normal physiological state", // plausible but false
        checksum = Guid.NewGuid().ToString() // helps look 'legit'
    });
}
public static EncryptedResponse GenerateFakeEncryptedPayload()
{
    // Generate pseudo—cipher bytes using Enumerable.Range
    byte[] fakeCipher = Enumerable.Range(0, 32)
        .Select(_ =>
        {
            var b = new byte[1];
            RandomNumberGenerator.Fill(b);
            return b[0];
        })
        .ToArray();
    byte[] fakeNonce = Enumerable.Range(0, 12)
        .Select(_ =>
        {
            var b = new byte[1];
            RandomNumberGenerator.Fill(b);
            return b[0];
        })
        .ToArray();
    byte[] fakeTag = Enumerable.Range(0, 16)
        .Select(_ =>
        {
            var b = new byte[1];
            RandomNumberGenerator.Fill(b);
            return b[0];
        })
        .ToArray();
    return new EncryptedResponse
    {
        EncryptedText = Convert.ToBase64String(fakeCipher),
        Nonce = Convert.ToBase64String(fakeNonce),
        Tag = Convert.ToBase64String(fakeTag)
    };
}
private static string GenerateFakeName()
{
    string GeneratePart(int length)
    {
        var rnd = RandomNumberGenerator.Create();
        return string.Concat(
            Enumerable.Range(0, length)
                .Select(i =>
                {
                    var buffer = new byte[1];
                    rnd.GetBytes(buffer);
```

```
                int seed = buffer[0];
                if (i % 2 == 0)
                    return _consonants[seed % _consonants.Length];
                else
                    return _vowels[seed % _vowels.Length];
            })
        );
    }
    string firstName = GeneratePart(5);
    string lastName = GeneratePart(7);
    string Capitalize(string word) =>
        char.ToUpper(word[0]) + word.Substring(1);
    return $"{Capitalize(firstName)} {Capitalize(lastName)}";
  }
}
```

ДОДАТОК Д

Програма Medical Gateway

```
using CulturalShare.Gateway.Configuration.Base;
using Microsoft.AspNetCore.Builder;
using Serilog.Core;
using System.Reflection;
namespace CulturalShare.Gateway.DependencyInjection;
public static class ApplicationConfigurationExtension
{
   public static WebApplicationBuilder InstallServices(this WebApplicationBuilder builder, Logger
logger, params Assembly[] assemblies)
   {
      var serviceInstallers = assemblies.SelectMany(x => x.DefinedTypes)
           .Where(x => IsAssignableToType<IServiceInstaller>(x))
           .Select(Activator.CreateInstance)
           .Cast<IServiceInstaller>();
      foreach (var serviceInstaller in serviceInstallers)
      {
         serviceInstaller.Install(builder, logger);
      }
      return builder;
   }
   private static bool IsAssignableToType<T>(TypeInfo type) =>
      typeof(T).IsAssignableFrom(type) && !type.IsInterface && !type.IsAbstract;
}
using CulturalShare.Foundation.AspNetCore.Extensions.Constants;
using CulturalShare.Foundation.Authorization.JwtServices;
using CulturalShare.Gateway.Configuration.Base;
using Google.Api;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Serilog.Core;
namespace CulturalShare.Gateway.Configuration;
public class ApplicationServiceInstaller : IServiceInstaller
{
   public void Install(WebApplicationBuilder builder, Logger logger)
   {
      builder.Services.AddScoped<IJwtBlacklistService, JwtBlacklistService>();
      builder.Services.AddHttpContextAccessor();
      builder.Services.AddHeaderPropagation(options =>
options.Headers.Add(LoggingConsts.CorrelationIdHeaderName));
      builder.Services.AddControllers();
      builder.Services.AddEndpointsApiExplorer();
      logger.Information($"{nameof(ApplicationServiceInstaller)} installed.");
   }
}
using AuthenticationProto;
using CulturalShare.Foundation.AspNetCore.Extensions.Helpers;
using CulturalShare.Gateway.Extensions;
using Google.Api;
```

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity.Data;
using Microsoft.AspNetCore.Mvc;
using PostsReadProto;
namespace CulturalShare.Gateway.Controllers;
[Route("api/[controller]")]
[ApiController]
public class AuthController : ControllerBase
{
   private readonly AuthenticationGrpcService.AuthenticationGrpcServiceClient _authClient;
   private readonly ILogger<AuthController> _logger;
   public AuthController(
      AuthenticationGrpcService.AuthenticationGrpcServiceClient authClient,
      ILogger<AuthController> logger)
   {
      _authClient = authClient;
      _logger = logger;
   }
   [HttpPost("SignIn")]
   public async Task<IActionResult> LoginAsync([FromBody] SignInRequest request,
CancellationToken cancellationToken)
   {
      _logger.LogDebug($"{nameof(LoginAsync)} request.");
      var headers = HttpHelper.CreateHeaderWithCorrelationId(HttpContext);
      var result = await _authClient.SignInAsync(request, headers, cancellationToken:
cancellationToken);
      return Ok(result);
   }
   [Authorize]
   [HttpPost("SignOut")]
   public async Task<IActionResult> RefreshTokenAsync([FromBody] SignOutRequest request,
CancellationToken cancellationToken)
   {
      _logger.LogDebug($"{nameof(RefreshTokenAsync)} request.");
      var headers = await this.CreateSecureHeaderWithCorrelationId(HttpContext);
      var result = await _authClient.SignOutAsync(request, headers, cancellationToken:
cancellationToken);
      return Ok(result);
   }
   [Authorize]
   [HttpPost("RefreshToken")]
   public async Task<IActionResult> RefreshTokenAsync([FromBody] RefreshTokenRequest
request, CancellationToken cancellationToken)
   {
      _logger.LogDebug($"{nameof(RefreshTokenAsync)} request.");
      var headers = await this.CreateSecureHeaderWithCorrelationId(HttpContext);
      var result = await _authClient.RefreshTokenAsync(request, headers, cancellationToken:
cancellationToken);
      return Ok(result);
   }
}
using CulturalShare.Foundation.Authorization.AuthenticationExtension;
```

```
using CulturalShare.Foundation.EntironmentHelper.EnvHelpers;
using CulturalShare.Gateway.Configuration.Base;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Serilog.Core;
namespace CulturalShare.Gateway.Configuration;
public class AuthenticationServiceInstaller : IServiceInstaller
{
   public void Install(WebApplicationBuilder builder, Logger logger)
   {
      var sortOutCredentialsHelper = new SortOutCredentialsHelper(builder.Configuration);
      var jwtSettings = sortOutCredentialsHelper.GetJwtServicesConfiguration();
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
               .AddJwtBearer(options =>
               {
                  JwtExtension.ConfigureJwtBearerOptions(options, jwtSettings);
               });
      builder.Services.AddAuthorization();
      logger.Information($"{nameof(AuthenticationServiceInstaller)} installed.");
   }
}
using AuthenticationProto;
using CulturalShare.Foundation.AspNetCore.Extensions.Extensions;
using Grpc.Core;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
namespace CulturalShare.Gateway.Extensions;
public static class ControllerExtension
{
   public static async Task<Metadata> CreateSecureHeaderWithCorrelationId(this ControllerBase
controllerBase, HttpContext httpContext)
   {
      var authHeaders = new Metadata().AddAuthHeader(GetAuthToken(httpContext));
      var headers = authHeaders.AddCorrelationIdHeader(httpContext);
      return headers;
   }
   private static string GetAuthToken(HttpContext httpContext)
   {
      if (httpContext == null || httpContext.Request == null)
      {
         throw new ArgumentNullException(nameof(httpContext), "HttpContext is null.");
      }
      var authHeader = httpContext.Request.Headers["Authorization"].FirstOrDefault();
      if (string.IsNullOrEmpty(authHeader) || !authHeader.StartsWith("Bearer ",
StringComparison.OrdinalIgnoreCase))
      {
         return null;
      }
      return authHeader.Substring("Bearer ".Length).Trim();
   }
}
```

```
using CulturalShare.Foundation.EntironmentHelper.EnvHelpers;
using CulturalShare.Gateway.Configuration.Base;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Serilog.Core;
using StackExchange.Redis;
namespace Common.Configuration;
public class DatabaseServiceInstaller : IServiceInstaller
{
   public void Install(WebApplicationBuilder builder, Logger logger)
   {
      var sortOutCredentialsHelper = new SortOutCredentialsHelper(builder.Configuration);
      builder.Services.AddSingleton<IConnectionMultiplexer>(provider =>
      {
         return
ConnectionMultiplexer.Connect(sortOutCredentialsHelper.GetRedisConnectionString());
      });
      logger.Information($"{nameof(DatabaseServiceInstaller)} installed.");
   }
}
using AuthenticationProto;
using CulturalShare.Foundation.EntironmentHelper.EnvHelpers;
using CulturalShare.Gateway.Configuration.Base;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Newtonsoft.Json;
using PostsReadProto;
using PostsWriteProto;
using Serilog.Core;
namespace CulturalShare.Gateway.Configuration;
public class GrpcClientServiceInstaller : IServiceInstaller
{
   public void Install(WebApplicationBuilder builder, Logger logger)
   {
      var sortOutCredentialsHelper = new SortOutCredentialsHelper(builder.Configuration);
      var grpcClientsUrlConfiguration =
sortOutCredentialsHelper.GetGrpcClientsUrlConfiguration();

builder.Services.AddGrpcClient<AuthenticationGrpcService.AuthenticationGrpcServiceClient>(op
tions =>
      {
         options.Address = new Uri(grpcClientsUrlConfiguration.AuthClientUrl);
      });
      builder.Services.AddGrpcClient<UserGrpcService.UserGrpcServiceClient>(options =>
      {
         options.Address = new Uri(grpcClientsUrlConfiguration.AuthClientUrl);
      });
      builder.Services.AddGrpcClient<PostsRead.PostsReadClient>(options =>
      {
         options.Address = new Uri(grpcClientsUrlConfiguration.PostReadClientUrl);
      });
      builder.Services.AddGrpcClient<PostsWrite.PostsWriteClient>(options =>
```

```
        {
            options.Address = new Uri(grpcClientsUrlConfiguration.PostWriteClientUrl);
        });
        logger.Information($"{nameof(GrpcClientServiceInstaller)} installed.
{JsonConvert.SerializeObject(grpcClientsUrlConfiguration)}");
    }
}
using CulturalShare.Gateway.Configuration.Base;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Serilog.Core;
namespace CulturalShare.Gateway.Configuration;
public class HealthCheckServiceInstaller : IServiceInstaller
{
    public void Install(WebApplicationBuilder builder, Logger logger)
    {
        builder.Services.AddHealthChecks();

        logger.Information($"{nameof(HealthCheckServiceInstaller)} installed.");
    }
}
using Microsoft.AspNetCore.Builder;
using Serilog.Core;
namespace CulturalShare.Gateway.Configuration.Base;
public interface IServiceInstaller
{
    void Install(WebApplicationBuilder builder, Logger logger);
}
using CulturalShare.Foundation.AspNetCore.Extensions.Extensions;
using CulturalShare.Gateway.Configuration.Base;
using Microsoft.AspNetCore.Builder;
using Serilog.Core;
namespace CulturalShare.Gateway.Configuration;
public class LoggingServiceInstaller : IServiceInstaller
{
    public void Install(WebApplicationBuilder builder, Logger logger)
    {
        builder.UseCustomSerilog(builder.Configuration);
        logger.Information($"{nameof(LoggingServiceInstaller)} installed.");
    }
}
using CulturalShare.Foundation.AspNetCore.Extensions.Extensions;
using CulturalShare.Gateway.Configuration.Base;
using CulturalShare.Gateway.DependencyInjection;
using HealthChecks.UI.Client;
using Microsoft.AspNetCore.Diagnostics.HealthChecks;
using Serilog;
var builder = WebApplication.CreateBuilder(args);
var logger = new LoggerConfiguration()
    .ReadFrom.Configuration(builder.Configuration)
    .CreateLogger();
builder.InstallServices(logger, typeof(IServiceInstaller).Assembly);
```

```
var app = builder.Build();
app.UseExceptionsHandler();
app.UseCorrelationIdMiddleware();
// Configure the HTTP request pipeline.
if (app.Environment.IsEnvironment("Test"))
{
   app.UseSwagger();
   app.UseSwaggerUI();
}
app.UseSecureHeaders();
app.UseHttpsRedirection();
app.UseAuthentication();
app.UseAuthorization();
app.MapHealthChecks("/_health", new HealthCheckOptions()
{
   ResponseWriter = UIResponseWriter.WriteHealthCheckUIResponse
});
app.UseHeaderPropagation();
app.MapControllers();
logger.Information($"Env:
{Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT")} Running App...");
app.Run();
logger.Information("App finished.");
using CulturalShare.Gateway.Configuration.Base;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.OpenApi.Models;
using Serilog.Core;
using System.Reflection;
namespace CulturalShare.Gateway.Configuration;
public class SwaggerServiceInstaller : IServiceInstaller
{
   public void Install(WebApplicationBuilder builder, Logger logger)
   {
      builder.Services.AddEndpointsApiExplorer();
      builder.Services.AddSwaggerGen();
      builder.Services.AddSwaggerGen(c =>
      {
         c.SwaggerDoc("v1", new OpenApiInfo
         {
            Title = "Medical Gateway Service API",
            Version = "v1",
         });
         // Set the comments path for the Swagger JSON and UI.
         var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
         var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
         c.IncludeXmlComments(xmlPath);
      });

      logger.Information($"{nameof(SwaggerServiceInstaller)} installed.");
   }
}
```

```
using AuthenticationProto;
using CulturalShare.Foundation.AspNetCore.Extensions.Helpers;
using CulturalShare.Gateway.Extensions;
using Microsoft.AspNetCore.Mvc;
namespace WebApi.Controllers;
[Route("api/[controller]")]
[ApiController]
public class UserController : ControllerBase
{
   private readonly UserGrpcService.UserGrpcServiceClient _userClient;
   private readonly ILogger<UserController> _logger;
   public UserController(
      UserGrpcService.UserGrpcServiceClient userClient,
      ILogger<UserController> logger)
   {
      _userClient = userClient;
      _logger = logger;
   }
   [HttpPost("CreateUser")]
   public async Task<IActionResult> CreateUserAsync([FromBody] CreateUserRequest request,
CancellationToken cancellationToken)
   {
      _logger.LogDebug($"{nameof(CreateUserAsync)} request.");
      var headers = HttpHelper.CreateHeaderWithCorrelationId(HttpContext);
      var result = await _userClient.CreateUserAsync(request, headers, cancellationToken:
cancellationToken);
      return Ok(result);
   }
   [HttpGet("GetUser")]
   public async Task<IActionResult> AllowUserAsync([FromBody] AllowUserRequest request,
CancellationToken cancellationToken)
   {
      _logger.LogDebug($"{nameof(AllowUserAsync)} request.");
      var headers = await this.CreateSecureHeaderWithCorrelationId(HttpContext);
      var result = await _userClient.AllowUserAsync(request, headers, cancellationToken:
cancellationToken);
      return Ok(result);
   }
}
```